MASSIS

– **M**ulti-**A**gent **S**ystem **S**imulation of **I**ndoor **S**cenarios–

Rafael Pax

rpax@ucm.es

Grado en Ingeniería en Informática

Facultad de Informática

Trabajo de Fin de Grado

Director :

Juan Pavón Mestras

jpavon@fdi.ucm.es

Madrid, Junio 2015

Authorization For Dissemination And Use

Autorizo a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor, tanto la propia memoria, como el código, la documentación y/o el software desarrollado.

Rafael Pax

Gracias por ayudarme con tus consejos,
por fortalecerme con tu apoyo,
por levantarme con tus ánimos,
por confiar en mí.

Gracias por alegrarme con tus risas,
por enamorarme con tu mirada,
por contagiarme tu locura,
por hacerme feliz.

Gracias por ser maravillosa, Pilar.

Acknowledgements

This work has been been supported by the Government of the Region of Madrid through the research programme MOSI-AGIL-CM (grant P2013/ICE-3019, co-funded by EU Structural Funds FSE and FEDER), and by the Spanish Ministry for Economy and Competitiveness, with the project Social Ambient Assisting Living - Methods (SociAAL) (grant TIN2011-28335-C02-01).

Resumen

Las pruebas de aplicaciones para entornos inteligentes son una tarea difícil. Requieren de la instalación de sensores y actuadores, los sistemas de comunicación, software de control y la participación de personas representando diferentes roles. Esto es costoso, tanto en tiempo como en sentido económico. Además, existen muchas situaciones que, por razones prácticas, resultan bastante complicadas de probar (situaciones de emergencia, por ejemplo). Las herramientas de simulación pueden servir de considerable ayuda para el desarrollo de entornos inteligentes. Uno de los aspectos más relevantes a tener en cuenta en este tipo de pruebas es el comportamiento humano y social de los individuos, cuando se simula la forma de cómo las personas interactúan con su entorno, incluyendo otros individuos. Si se utiliza un framework de simulación multiagente para estos propósitos, debe constar de un modelo claro de agente, cuyos métodos de razonamiento puedan ser diseñados desde un nivel de abstracción más alto, que pueda transformarse en una implementación de forma sencilla.

Éste es uno de los propósitos principales de MASSIS (Multi-agent system simulation of InDoor Scenarios), un framework de simulación multiagente eficiente que permite el modelado y la simulación de los procesos de toma de decisiones de los agentes en múltiples situaciones en el dominio de espacios interiores. Extiende las capacidades de SweetHome3D con ciertos plugins que permitan definir el comportamiento de los agentes en el entorno de la simulación. Otras funcionalidades que ofrece MASSIS son las visualizaciones 2D y 3D de la simulación, la capacidad de guardar las simulaciones para su posterior reproducción y análisis.

Palabras clave

- Modelado basado en agentes
- Modelo de decisión de agentes
- Simulación de multitudes
- Escenarios de interiores
- Entornos inteligentes
- Framework de simulación

Abstract

Testing applications for smart environments is a difficult task. It requires the installation of sensors and actuators, the communications and the software for the control system, and the participation of people playing different scenarios. This is costly, both in economic sense as well as in time. Also, there are some situations that cannot be tested for practical reasons (such as emergencies).

The use of simulation tools that provide some support for the development of smart environment applications is interesting, at least for these reasons. One of the most relevant aspects to be considered in this kind of tests is the human and social behavior of individuals when simulating how people interact with their environment, including other individuals. If the simulation framework has to be used for different purposes and by other developers, it should have a clear agent model, with some support for the design at a higher level of abstraction that can be easily translated to an implementation.

This is the main purpose of MASSIS (Multi-agent system simulation of InDoor Scenarios), an efficient framework for modeling and simulation of the decision-making process of agents in multiple situations in indoor scenarios domain. It extends the SweetHome3D environment with plugins for linking agent's behavior in the simulation. Other functionality provided by MASSIS is the ability to visualize the simulation in 2D and 3D, and a rich log capability, which can be the basis for further analysis of the scenarios.

Keywords

- Agent-based modeling
- Agent decision model
- Crowd simulation
- Indoor scenarios
- Ambient Intelligence
- Simulation Framework

Table of Contents

List of Figures

Code Listings

Appendices

List of Abbreviations

AAA (video game)	Classification term used for games with the highest development budgets
ABS	Agent Based Simulation
API	Application Program Interface
BOD	Behavior Oriented Design
BUP-QT	Bottom-Up Propagation QuadTree
DES	Discrete Event Simulation
F.E.A.R	First Encounter Assault Recon
GOAP	Goal Oriented Action Planning
JSON	JavaScript Simple Object Notation
JTS	Java Topology Suite
MASON	Multi-Agent Simulator Of Neighborhoods / Networks
MASSIS	Multi-Agent System Simulation of Indoor Scenarios
NPC	Non-Player Character
POSH	Parallel-Rooted, Ordered Slip-Stack Hierarchical
SD	System Dynamics
STRIPS	Stanford Research Institute Problem Solver
UDK	Unreal Development Kit
UE2	UnrealEngine2RuntimeDemo
UT2004	Unreal Tournament 2004

1 Introduction

Let's take flight simulation as an example. If you're trying to train a pilot, you can simulate almost the whole course. You don't have to get an airplane until late in the process.

Roy Romer

1.1 Motivation Of The Work

A simulation model is basically a set of rules governing a system, which determine how it changes over time. Unlike other types of models, (e.g. analytical), a simulation model is not solved; *it is executed.*

The changes in the system can be observed over time, providing a valuable insight into the system dynamics, instead of obtaining the system's output for a certain given input. Of course, the real world is extremely complex, and a simulation model can only be an approximation of the real system. Therefore, there is a tradeoff between the realism of the simulation, the ability to model complex behaviors of the actors, and the control and monitoring of what happens during the simulation. In addition, the simulation also presents other interesting challenges such as the scalability of the models, or the efficiency of the simulation platform when modeling non-trivial systems.

Rather than considering simulation as a *decision-making tool*, simulation should be considered as *a tool to support decision making*. Considering this, making simulations about a real world system without testing it in real life can help the development of several applications, making this process faster and cheaper.

In the last two decades, agent-based modeling is being increasingly used for the study of complex systems in various fields of so-

cial sciences, as well as support for decision making. Entities belonging to social systems can be modeled as autonomous agents interacting with an environment. These models can be simulated to analyze the behavior shown by the system, in different scenarios and configurations.

An interesting social system scenario is an smart environment, where people are surrounded by different types of sensors, actuators and computing components designed to make their life easier. Testing applications in this type of environments can be difficult. Modeling an smart environment as a multi agent system would make easier the information retrieval about how prototypes will behave without actually testing in real life, helping to reduce time and costs.

This work has been developed under the context of the MOSI_AGIL research project of UCM-GRASIA, in collaboration with research groups of URJC and UPM universities. The goal of the project is to study the behavior of crowds of people under different scenarios. The project started using the Ubiksim tool [1], which was developed in a national project by Univ. Murcia in collaboration with UCM. After using this tool during the first two months of the project, several limitations where identified on Ubiksim, which made difficult to implement many of the new ideas that emerged during the beginning of the project. For that reason, and with the motivation to create something more extensible and more efficient, it was decided to redesign and implement a new platform, MASSIS. This new framework is going to be used in the next phases of the MOSI-AGIL project, with some of the lines of research that are presented in section 5.2.

One of the outcomes of the work done has resulted in two papers for international conferences, which are copied in an annex to this document:

- Rafael Pax, Juan Pavón: Agent-based Simulation of Crowds in Indoor Scenarios. 9th International Symposium on Intelligent Distributed Computing (IDC'2015), Guimaraes (Portugal), 7-9 oct. 2015 (accepted for oral presentation and full publication)
- Rafael Pax, Juan Pavón: Multi-agent system simulation of InDoor Scenarios. 9th International Workshop on Multi-Agent Systems and Simulation (MAS&S'15). Lodz, Poland, September 13-16, 2015 (submitted, waiting for acceptance)

1.2 Objectives

The main purpose of the project is to develop a framework for multi agent modeling and simulation of indoor scenarios. This framework has to support the specification of rich and heterogeneous agent behaviors, without compromising the scalability of the simulations. The framework has to be component-based in order to facilitate its extensibility.

This involves the achievement of the following objectives:

- A fast and easy way for modeling indoor environments.
- A model for specifying different types of behaviors, which allows the definition of the interaction of the agent with its environment.
- A simulation framework, capable of
  - Handling several agents, each one with its own behavior, keeping in mind scalability and efficiency.
  - Displaying in real time the simulation state from different perspectives, such as 3D and 2D views.
  - Saving the changes that the initial simulation state had over time, in an standardized and open format, allowing the analysis of the results later without restricting this analysis to a particular language or platform. Also, it should allow playback of previously stored simulations.
  - Being extensible, flexible and as far as possible, multipurpose. For this, both the platform and its components shall have well defined interfaces, and will be provided as open source.

1.3 Methodological Approach

Art and science have their meeting point in method.
Edward G. Bulwer-Lytton

For the development of this project, Agile software development principles were followed. Due to the small size of the team (1 person), and the nature of this project, an adaptation of these principles was needed.

1.3.1 Conception Phase

The project started with an analysis phase. This phase was necessary because the team had no domain-specific knowledge. A basic understanding of the domain was needed in order to be capable of determine potential problems and needs. In this phase a very simple process was followed:

- Meeting with the advisor once a week.
- Definition of the goals that should be accomplished for the next week.
- Writing down new ideas, and defining project objectives.

1.3.2 Agile Development

The development of the project started on October, 2014. It was decided to use a simplified agile methodology, adapted to take into account that there were only two main participants: the student and the advisor. In order to facilitate high interaction between them, two communication mechanisms were used: emails and meetings.

Two E-mails per day, meeting whenever possible

In order to keep constant communication between the student and the advisor, and monitoring of project progress, every day in the morning, an e-mail was written to the advisor, explaining the expected goals to be accomplished that day. At the end of the day,

an email containing a brief summary of the achievements of the day, the problems faced, and occasionally, a new idea to consider.

Face-to-face communication is one of the pillars of agile development. Whenever the team and the advisor found time to meet, they met. There was not a fixed day, but usually was once a week.

Fast sprints and short iterations

Following Agile Principles require frequent deliveries of prototypes of useful products. But one of the main problems of a one man team is that few tasks can be done in parallel. Focusing on a single sprint at a time, and limiting the amount of work-in-progress provides a better project development. For that reason, the iterations were composed of different sprint types, each one lasting one week. Different types of sprints were designed:

- Literature review sprints: These sprints were intended for learning about new frameworks, reading and summarizing research papers, in order to make the team capable of making more difficult tasks. During this sprints, it was very common the emergence of new ideas. These new ideas were classified with a priority. These types of sprints were made only if the results of the previous sprints were satisfactory.
- Implementation sprints: These sprints focused on the implementation of the highest priority feature to implement at the moment. They were focused primarily on implementing the tasks defined before.
- Testing sprints: They were focused on testing the functionality of the system, bug fixing and performance analysis.
- System architecture review sprints: After implementation and testing of the new feature/goal, the overall system architecture is revisited, in order to maintain coherence. The system architecture review sprints had at least one face-to-face meeting with the advisor.

Figure 1 shows the development work flow of the iterations.

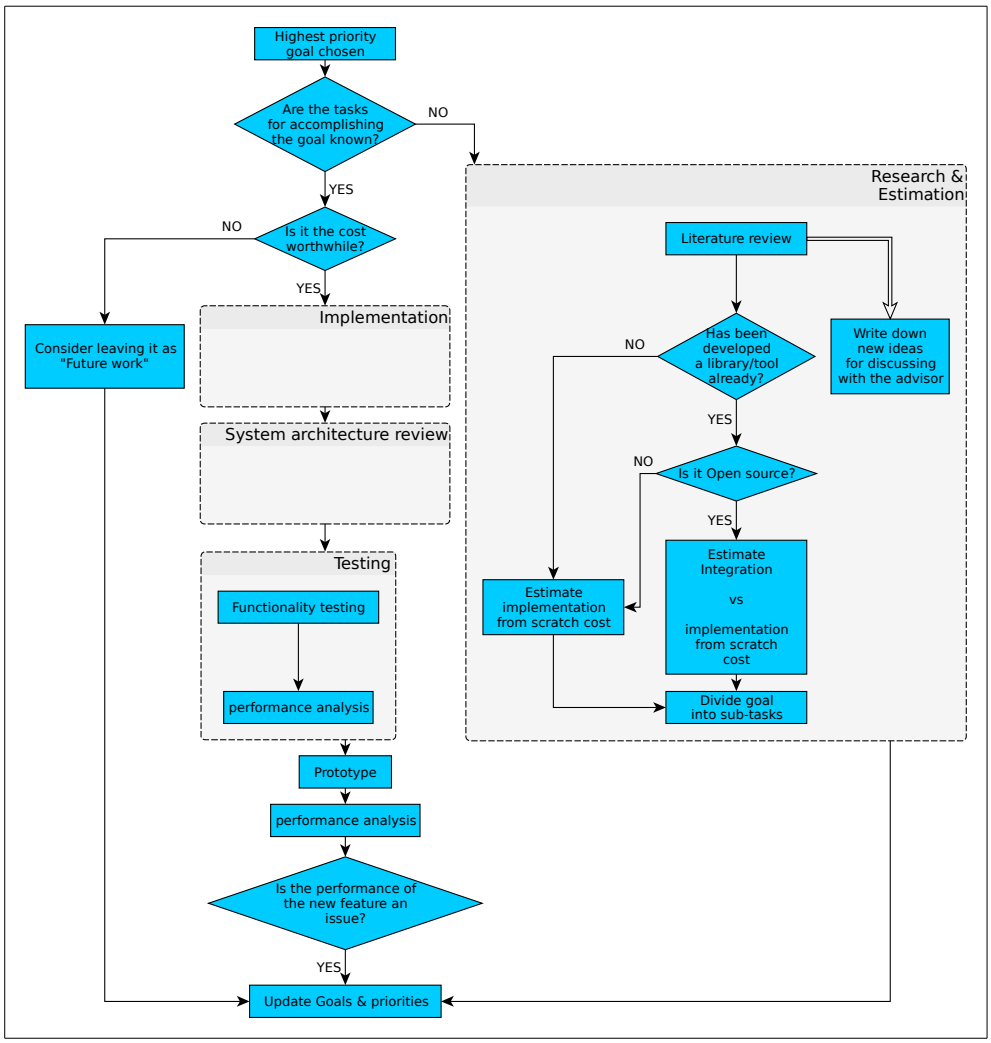

Figure 1: Development methodology

1.4 Document Organization

The rest of the document is organized as follows:

- **Section 2** reviews the state of the art, where the most relevant aspects, both theoretical and technological are discussed. In its first section, the main tools are reviewed, along with their different characteristics. The second delves into the simulation of indoor scenarios, highlighting the most important problems. Finally, the potential of multi-agent tools is shown, and the need for a tool like MASSIS is explained.
- **Section 3** Introduces the MASSIS (Multi-agent System Simulation of InDoor Scenarios)
  architecture and components. Special attention is given to the definition of the behavior of humans under different situations,
  which includes the process for decision making of the agents. Other relevant aspects to model are interactions among agents and with their environment, the events on the environment, and the precise representation of the building, logging capabilities and visualization.
- **Section 4** Explains how to get started with MASSIS, the installation, environment creation, behavior modeling and simulation.
- **Section 5** Presents the conclusions and future work

2 Simulation Of Smart Environments

The business of art is to reveal the relation between man and his environment.

D. H. Lawrence

Smart Environments[2] can be defined as *"a region of the real world that is extensively equipped with sensors, actuators and computing components"* [3]. That is, a smart environment is aware of what occurs within it and its surroundings, adapting how it behaves to achieve certain objectives (e.g. making easier the lives of the inhabitants of a building). Previous works [4] have proposed the approach of viewing smart homes as intelligent agents, that perceive their environment through the use of sensors, and can act upon the environment through the use of actuators, agents that constantly adapt their behavior to the behavior of the environment itself.

Besides of the smart environment design, one of the main problems that arises during the development of these kind of environments is their *heterogeneity*: the entities which inhabit the environment may be very different to each other, not known a priori and *dynamic*: may change over time.

Although there have been several researches on this topic, such as [5]–[7], testing applications for smart environments continues being a difficult task. It requires the installation of sensors and actuators, the communications and the software for the control system, and the participation of people who have to play the different scenarios. This is costly, both in economic sense as well as in time. Also, there are some situations that cannot be tested for practical reasons (e.g., a fire, people accidents).

Furthermore, from the point of view of the developers, who are used to iterative processes, it is difficult to repeat the tests if they have to perform these with persons. At least for these reasons is interesting to use simulation tools that provide some support for the development of smart environment applications. A relevant aspect to be considered in this kind of tests is the modeling of the behavior of humans under different situations. The behavior for these scenarios requires at least the following: interactions among agents, with the environment, and the process for decision making.

Although many agent-based modeling and simulation tools exist, framework, not specifically oriented to smart environments.

2.1 Agent-based Modeling And Simulation Tools

Nowadays, there are many software tools for implementing agent-based models. Although many of them are of general purpose, the selection of one or another depends on many factors, such as the scope and objective of the model to develop, the execution platform, documentation and ease of use, or the reusability of the code.

One of the most influential software packages, which has served as inspiration to current platforms is Swarm [8], which began in 1994. The most widely used platforms are based on the Swarm philosophy, which is oriented towards to the *framework and library* paradigm. A framework that defines the concepts of agent-based modeling, including also the libraries needed for implementing the concepts proposed according to the framework.

In general, simulation platforms follow the OO paradigm, where the framework defines the object in charge of building and controlling the simulation, as well as the objects responsible for the graphic elements management and the representation of the execution results, along with the scheduler system, which controls the execution of the events that trigger the methods defining the behavior of the agents.

Repast[9] has been one of the Swarm based frameworks has been more successful. It started at the University of Chicago, being oriented to the social sciences domain. The first versions of Repast facilitated the designing of models for users without an extensive knowledge of software development, but the fundamentals of object-oriented programming, and knowledge about the Java programming language were required. Today, Repast continues its development at Argonne National Laboratory. The latest version, Repast Symphony [10], has evolved significantly.

MASON[11], developed at George Mason University, appeared shortly after Repast, as a computationally efficient alternative. Although MASON is also based on the Swarm model, it shares with Repast many of aspects mentioned before. MASON increases the independence of the application domain of the models and enhances substantially those characteristics necessary in very demanding computer models, such as hardware independence, independence interfaces display and serialization.

Netlogo [12] is another one of the most widespread platforms. It was developed at Northwestern University. Unlike Repast or MASON, it is based on its own high-level language (based on Logo, a Lisp dialect). One of the differences that Netlogo has over other libraries is that Netlogo is well documented and its library has many examples. Also, the programming style of Netlogo, which provides many primitives and simple construction of graphical interfaces, makes easier the learning process of the platform, especially for users without deep training in software development. Although it does not present the advantages of modularity and reuse of code that platforms based on more general-purpose languages, the ease of use and the ability to generate and share the models as executable applets in any browser that Netlogo has, has made it one of the tools most used for simple models.

2.2 Simulation Of Indoor Scenarios

Several tools for simulation and design of how people behave in indoor scenarios exist (many of them commercial), such as [13]–[18], among many others. They focus on scalability issues derived from the management of a large number of agents in real time, specially when considering their visualization or the way the agents find their way while avoiding obstacles and other agents [19].

Specially in the last years, focus has been on very large numbers of agents. Different techniques have been proposed to cope with the scalability issues, by relying on specific assumptions of the problem under study. This has an effect on limitation of the flexibility of agents' behavior, which is quite homogeneous in most of the cases.

Although they are appropriate to simulate specific scenarios, it is important to consider the human and social behavior of individuals when simulating how people interact with their environment, including other individuals. Other works have better addressed the specification of the agent behavior, such as [1], [20]–[25].

However, they have not sufficiently taken into account the methodological aspects for a design process when developing the agents' behavior. This is relevant when the simulation framework has to be used for different purposes and by other developers. In those cases, there is a need for a clearer agent model, with some support for the design at a higher level of abstraction that can be easily translated to an implementation.

2.3 Conclusions

Agent-based modeling provides many advantages over other modeling paradigms. In general, the process of abstracting the details of the target system and implementing them unambiguously on a computer is much more direct than other methods of abstrac-

tion. Therefore, the model is more transparent, which facilitates the understanding of the hypotheses assumed and the inclusion of knowledge of domain experts.

The result of this ease of abstraction is reflected in many ways. For example, the variety in behavior of agents is enormous. Agent-based modeling allows to consider the effect of agents having limited rationality, or agents with ability to learn; from classic probabilistic mechanisms (such as reinforcement learning), to complex models from cognitive psychology (e.g endorsement systems). Or even one step further: The agents could create their own models of the world they perceive. The options are almost limitless. The relaxation of the assumptions of representative agents, having the ability to interact with the entire population, and an optimized behavior of a utility function (very common in many social sciences) is done almost directly using this paradigm.

As have been stated in sections 2.1 and 2.2, there is a gap between general purpose libraries and specialized tools: Some are too open, while others are too specialized, or they do not fully integrate design, modeling and simulation. Taking this into account, this work proposes an agent-based model for indoor scenarios where both performance and flexibility in the behavior of the entities are sought. Agents are specified and managed individually, but the effects of the crowd are taken into account by several methods that take advantage of characteristics of the indoor domain in order to cope with the efficiency and scalability issues in the processing of their movements and their visualization. At the same time, some alternative reasoning mechanisms are provided for each agent in order to allow modeling of rich and heterogeneous behaviors.

Our approach has been to adopt an agent-based modeling framework, Mason, and provide on this several components as a kind of plugins that facilitate the modeling and simulation of indoor scenarios. Mason has been chosen because it can be easily integrated as a Java library, it clearly decouples model from views, it is light and has an efficient scheduler. The component-based architecture of Mason is shown in the next chapter.

3 MASSIS Overview

A complex system that works is invariably found to have evolved from a simple system that worked.
John Gall

MASSIS is a simulation framework for scenarios in indoor environments, allowing to design spaces, and specifying the behavior of the elements and people in them. These behaviors specifications may vary substantially: From a simple presence detector to human behavior. It is capable of supporting thousands of agents, each one with an specific behavior. The behavior specification is done outside the simulation platform. Although currently MASSIS provides only one behavior model (the POSH model, see section 3.1.3), others can be integrated, but this is left for further study.

The simulation progress can be visualized in 3D, from different perspectives, or in 2D. The 2D visualization library is based on layers (the elements of each layer are drawn on top of the previous layer), making easier the development of a new type of visualization for specific purposes. Also, it allows to save the simulation changes, recording each agent state in every step of the simulation. These changes are saved in an open and independent format (JSON), allowing the analysis of the results from any other platform and language.

Both MASSIS and its components are open source, allowing the extension of its functionality by third parties.

3.1 Framework

Don't wait for inspiration;
create a framework for it.

Jocelyn K. Glei

MASSIS has a component-based architecture, where some part of the infrastructure is well-proven open source software. Above of the core infrastructure there is an agent-based framework where flexible agent behavior types can be specified, as well as their interactions and the elements of the environment. Fig 2 shows the main components of the framework.

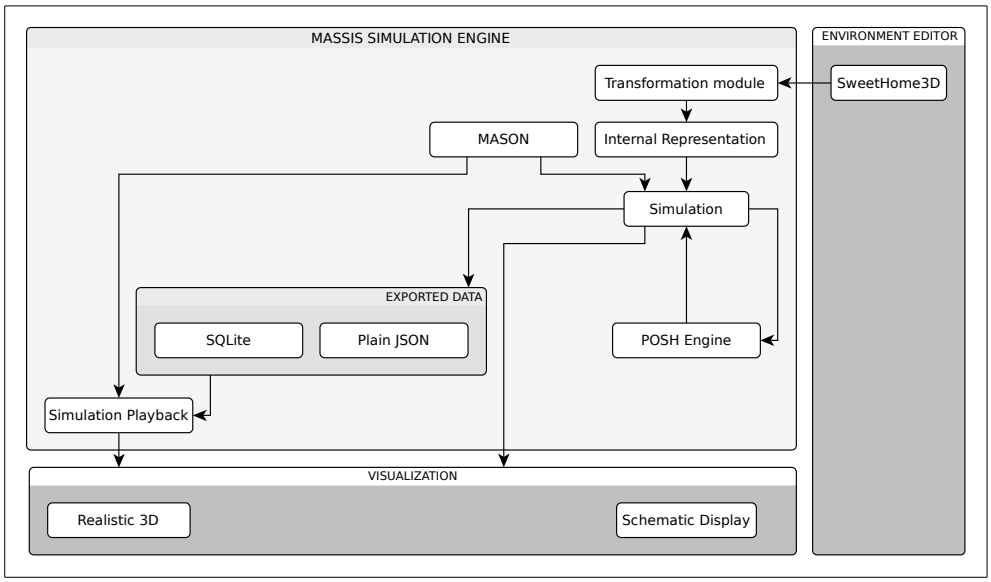

Figure 2: MASSIS' Framework Overview

3.1.1 SweetHome3D

The environment is defined using SweetHome3D[26], a well known package that is used to model all components involved in an indoor environment, such as walls, doors, stairs, people, etc. It is s a free building design software application that allows users to create 3D houses in an easy way, with 3D and 2D views. It also allows

the decoration of the interior and the exterior of the building with a high level of detail: 3D objects can be imported as furniture, and they can be arranged for creating a virtual environment, which appearance can be improved adding multiple textures and light points (Fig. 3). It also has useful display options, allowing filtering the elements (individually or by level), transparency of the walls (Figures , etc.

Figure 3: House designed with Sweet Home 3D.

It also allows extensibility via plugins. MASSIS takes advantage of this feature of SweetHome3D, and, in order to facilitate more flexibility of the characterization of the physical elements, a plugin to specify the characteristics of the elements was developed. With this plugin, the elements of the building designed with SweetHome3D can hold certain information in order to define its behavior. For instance, in the case of sensors and actuators, they will be reactive agents, with a simple behavior and attributes. In the case of people, some physical characteristics can be defined such as weight, speed, but also some inherent attributes of the person, like knowledge about the environment, reasoning capabilities, etc. and a link to their behavior. The types of behaviors are defined as components.

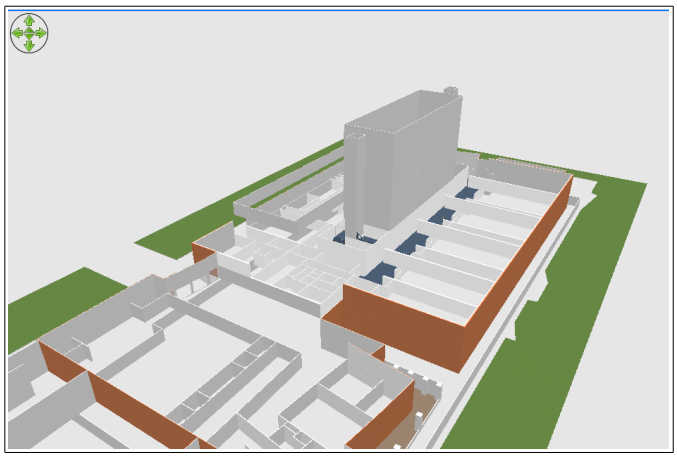

Figure 4: SweetHome 3D view (1)

When the building is created, the SweetHome3D representation of the building is transformed inside the simulation engine, which adapts it to the internal representation that is managed by MASSIS, in order to to support the efficiency of algorithms implementation, that wouldn't be possible otherwise; SweetHome3D is a home-design software, not a simulation framework.

Figure 5: SweetHome 3D view (2)

Although perhaps, it would have been easier to modify the source code of SweetHome3D (it is open-source) instead of developing plugins for the integration with MASSIS, it was not done to maintain compatibility between different versions of this software. In fact, during the development of project, MASSIS worked with versions 4.4, 4.5 and 4.6. In this way, every time SweetHome3D makes a new release benefits MASSIS: new features, bug correc-

tions, etc. benefits directly to MASSIS without having to make any changes to the framework.

3.1.2 MASON

The simulation engine of MASSIS is MASON[11], a lightweight multi-purpose agent-based simulation library, used as simulation engine. It has been chosen because it provides a good support for agent-based simulation platform, and outstands due to its efficiency.

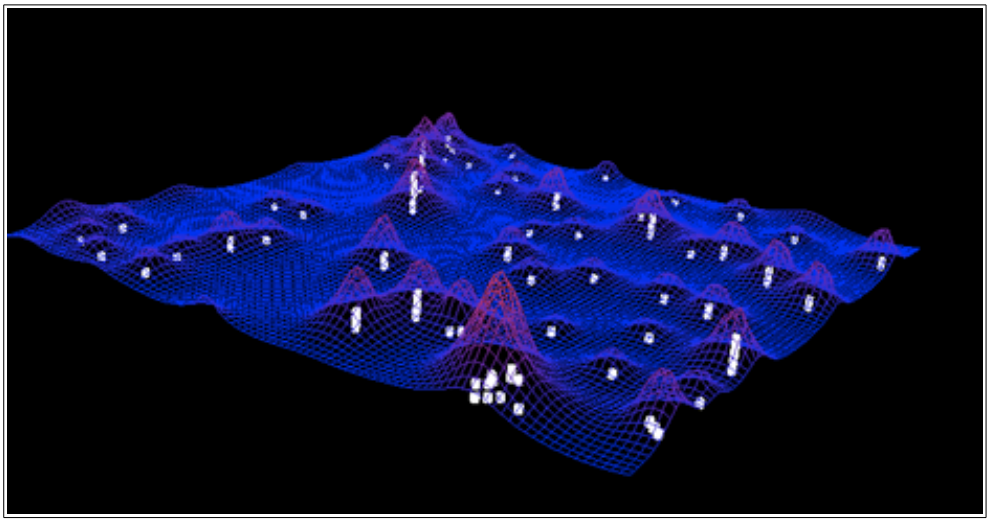

Figure 6: Example of MASON 3D Display

MASON clearly separates the model from the visualization, which implies that the model can be executed without any type of visualization. MASON
supports multiple types of visualization frames, both in 2D and 3D, that can be added, changed or removed easily. It also has support for saving the simulation state.

MASON Model

MASON's model consists of a discrete event schedule, on which agents can be scheduled for receiving its corresponding call in the future. It also contains several containers for handling space elements, called *fields*. They can be networks, continuous space or grids.

MASON Visualization

MASON provides tools for visualizing the simulation models in 2D and 3D (Figures 18 and 17). Visualization is wrapped in a *GUIState*, that contains a *controller*. This controller (usually a window) manages the 3D and/or 2D visualizations.

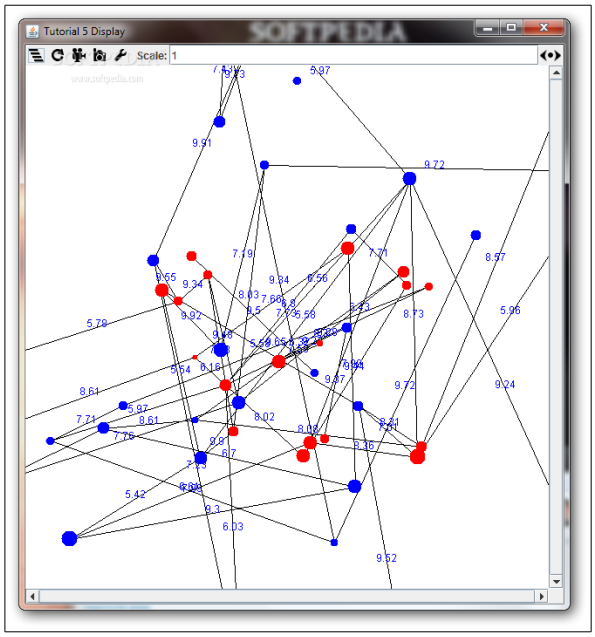

Figure 7: MASON 2D Display

Multiple displays can be defined, using *Field Portrayals*, which visualize the data stored in the model's fields. It also allows to "inspect" the model by selecting objects from the GUI, retrieving and showing information from the model.

MASON Utilities

MASON has several utilities that act as a swiss army knife when comes to designing a model, such as random number generators, different object collections, Java Beans Properties inspectors, GUI widgets, pictures and video generators and chart generators. MASSIS does not use MASON visualization portrayals, as the internal representation of MASSIS does not fit well with MASON visualization model. Despite this, thanks to how well the different components of the visualization controllers are structured, MASSIS display can run on top of MASON's console. Figure 8 shows the integration between MASSIS and MASON visualization.

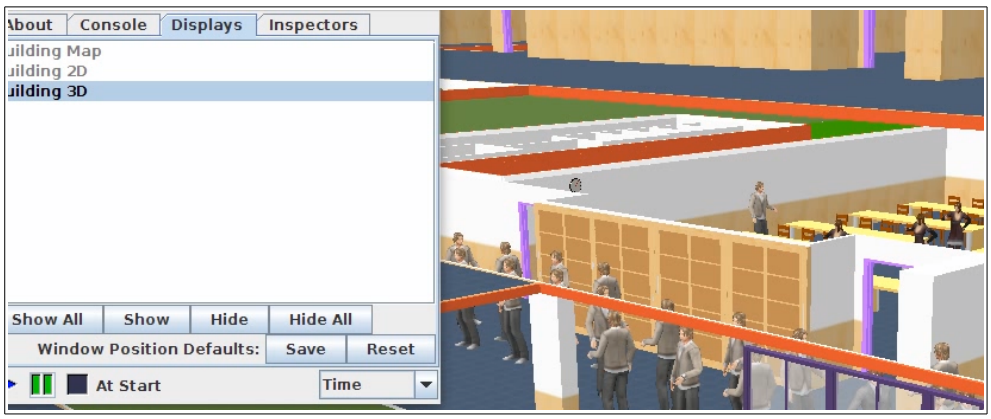

Figure 8: MASSIS and MASON visualization integration.

3.1.3 Pogamut's POSH Engine

Pogamut [27] is an open source platform intended for a fast development of the behavior for virtual agents in videogames. It acts as a middleware, allowing the management of virtual agents in environments that are provided by different game engines, such as Unreal Tournament 2004 (UT2004) [28], UnrealEngine2RuntimeDemo (UE2) [29], Unreal Development Kit (UDK) [30] and DEFCON[31].

It provides a Java API for controlling the virtual agents via Netbeans, with the objective of simplifying the physical elements of the agent, allowing the developer to focus on the AI part.

The interesting part from the MASSIS' point of view is that one of the modules developed in the Pogamut Framework was a new version of a POSH[1] engine, written entirely in Java. With some modifications in order to adapt it to MASSIS architecture, Pogamut POSH Engine works seamlessly in MASSIS (More information about POSH can be found on Section 3.6.2).

[1] Pogamut's POSH Engine is based on SPOSH (Strict-POSH) instead of POSH, but for simplicity, this paper will refer only to POSH.

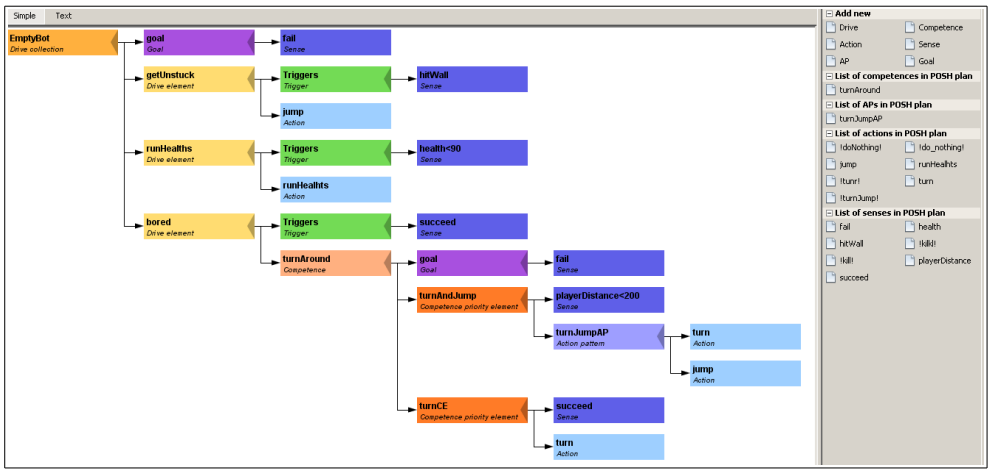

Figure 9: Integrated POSH Plan editor in Netbeans

In addition to this, the set of plugins for Netbeans developed by the Pogamut team include a POSH plan editor (Fig. 9). This plan editor helps with the AI development, allowing the parallel development of a higher-level abstraction AI and program code.

3.1.4 Saving & Loading Simulations

Also, the simulation events can be logged in JSON format, as a single zipped file or in a SQLite database for further analysis.

Once a simulation is performed, the exported data can be used to playback all events that have occurred during the execution of the simulation, i.e., the agents will behave in the same way they did during the simulation.

```
{
    "velocity": { "x": 32,"y": 58},
    "visionRadio": 300,
    "maxforce": 10,"maxspeed": 15,
    "properties": {"steering.separation": 70,...},
    "locationState":{
    "angle": 0.7853982,"floorId": 8,
    "centerX": 4975.2285,"centerY": 4108.2695,...},
    "id": 3673
}
```

Listing 1: Example of an agent's saved state

This is interesting to allow the users to review the simulation when analyzing what has happened.

3.1.5 Visualization

All the changes made in the environment are reflected in real time by 3D (Figure 10) and 2D displays (Figure 11).

Figure 10: Simulation 3D view

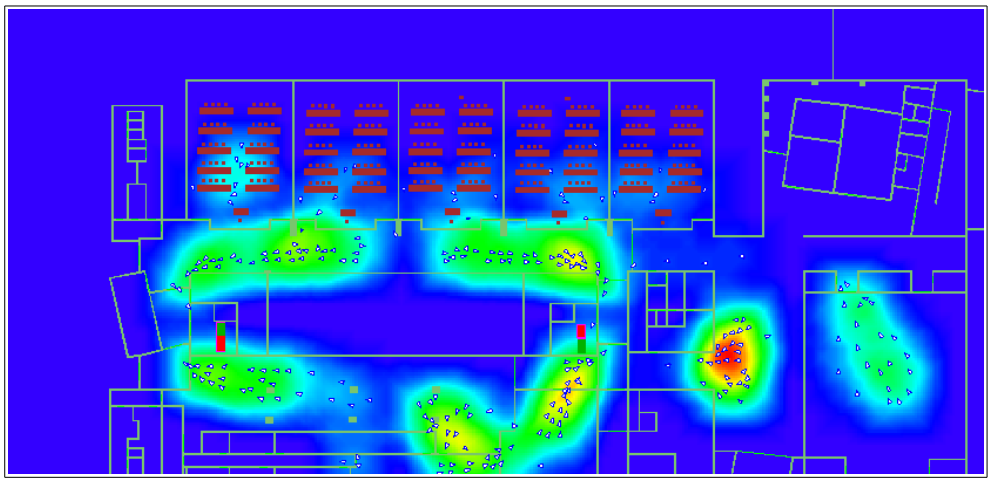

Figure 11: Simulation 2D view example : Crowd Density

Although 3D display is more realistic, the 2D view is useful for analysis and debugging. Also, the 2D visualization API allows the creation of user-defined layers in order to filter the different elements involved in the simulation.

3.1.6 StraightEdge

Of the open source libraries that are used in MASSIS, StraightEdge[32] deserves an special mention. It is a powerful polygon library, that provides great functionality to MASSIS engine.

It provides several geometric utilities, path-finding , field of vision and utility classes for converting StraightEdge polygons to Java Topology Suite (JTS)[33] polygons, allowing to perform more complex geometric operations, such as polygon flattening and shrinking, intersections and unions.

Several modules of StraightEdge were extended in order to improve their efficiency (Pathfinding) or to add new functionalities (e.g. vector operations or SweetHome3D polygon transformations).

3.2 Building Representation

Everybody at the party is a many sided polygon.

Nonagon, They Might Be Giants.

How the environment is represented is one of the key elements in any simulation framework, because this representation is the basis of all the simulation system. It is very important to choose an appropriate way to model the environment depending on the domain.

For example, an environment can be modeled as a continuous space, a grid, or a graph. Or even a combination of these. The underlying structures chosen to model the environments depend on the level of desired in the application.

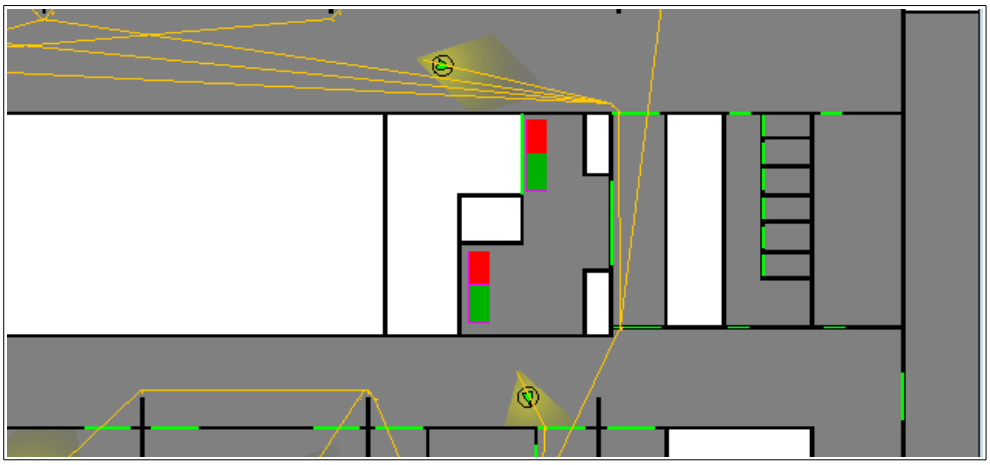

Figure 12: In MASSIS, almost every element is a polygon.

It is very common in the indoor scenario domain that elements are very close from each other. Agents are not the only ones considered here; tables, chairs, cabinets, sinks, even flowerpots or radiators.

Therefore, MASSIS opts for a representation of the building in which the space is not discretized in cells that are occupied by one

element or another. Instead, each element has a position in three dimensions: *(x, y, building-floor)*, and a polygon that represents it.

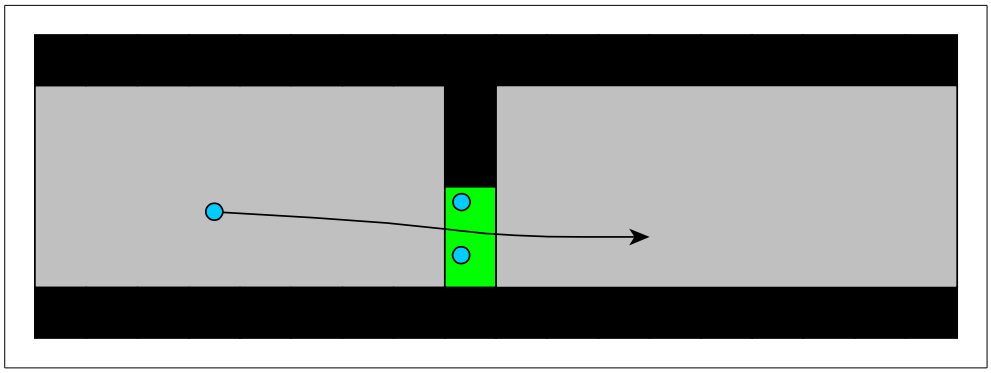

Figure 13: Movement in a continuous space model

This can be seen in Figure 12 , where everything is represented in a polygonal way: agents (green arrows enclosed by circles), doors (thin green rectangles), walls (black rectangles),rooms (gray areas), stairs,(contiguous red and green rectangles), vision areas,etc.

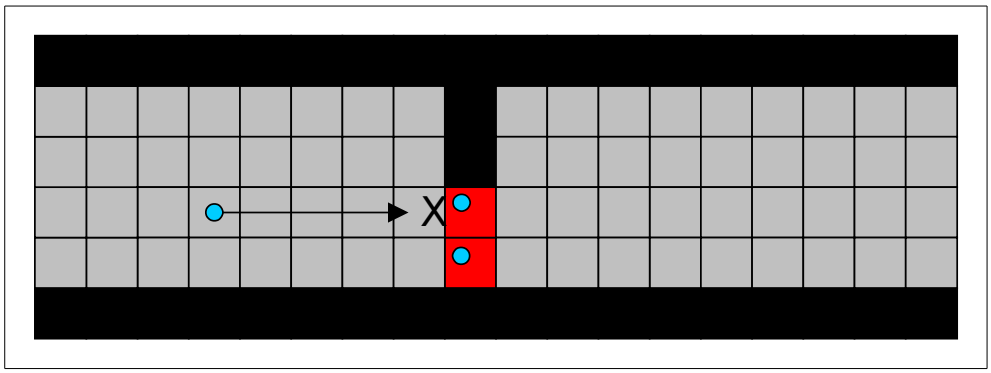

Figure 14: Movement in a low-resolution grid
In indoor scenarios, the use of a low-resolution grid may cause strange behaviors

An example that illustrates this need for a model of this type are doors. The width of the doors is a crucial factor in determining several aspects of the model, such as crowd congestion or pathfinding. The discretization level necessary to achieve the same results using uniform grids in a building with several floors with thousands of agents is somewhat prohibitive.

As it will be explained in subsequent sections, the use of points with particular data structures to manage them efficiently, rather than uniform grids has many advantages. One of them is shown in Figures 13 and 14 .

3.3 Path Finding

If you find a path with no obstacles, it probably doesn't lead anywhere.

Frank A. Clark

Pathfinding is one of the issues that has more impact when simulating crowd behaviors. Some models treat the crowd as a single entity or a group in order to simplify the number of calculations, such as in [34]–[36]. However, MASSIS, as it has been stated in the introduction, has as objective to support flexibility in agent behavior, therefore the path finding model is implemented individually for each agent, but taking advantages of some assumptions from the problem domain in order to gain in efficiency.

As it is explained in previous sections, the building model is represented on a continuous space. Therefore, it makes sense to follow this kind of design when developing a pathfinding module. If obstacles are represented as polygons, building a path through a visibility graph is a good approach.

3.3.1 Visibility Graph

A Visibility Graph is a graph whose nodes correspond to geometric components, such as vertices or edges, and the nodes of this graph are connected only if there is not any obstacle intersecting the segment between those two points. Each edge of this graph represents a *visible* connection between those two points (They can see each other). In Figure 26 shows an example of a visibility graph.

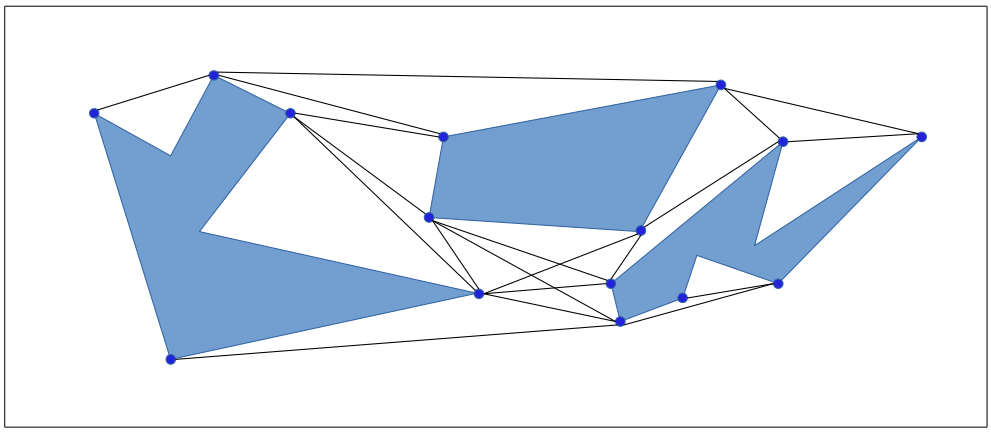

Figure 15: Visibility graph

As all the elements present in the simulation have a polygon associated to them, this representation of the elements is very handy for the obstacles preprocessing step. This preprocessing step is done before the simulation starts, in order to gain computational speed during the simulation.

The main reason for this preprocessing step is that some components of the building, such as walls, or stairs, are never going to move, so there's no reason to recompute the visibility graph of the building every time a path is requested. The preprocessing step consists basically of the following parts:

Obstacle merging

As MASSIS is intended for indoor scenarios simulations, some characteristics of this domain are exploited, such as the high frequency of wall intersection.

Usually the end of a wall is connected with another. In Sweet-Home 3D representations, each wall is considered as an individual polygon. This is very useful for the building design stage, but when the number of polygons becomes important, it could become a problem. The way the walls intersect makes possible the reduction of the number of edges. (See Fig.16)

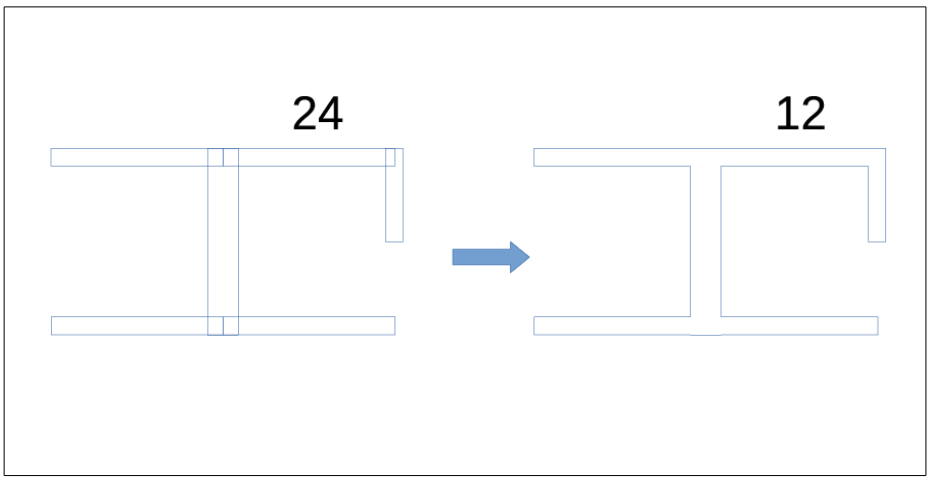

Figure 16: Reduction of number of edges.

Obstacle flattening

The geometry of the obstacles is expanded by an amount proportional to the bounding radius of the agents. This process is done in order to make generated paths more realistic, because the path is generated with the vertices of the expanded obstacle, not from the obstacle itself, so the path that the agent tries to follow is slightly separated from the obstacles vertices and edges. Fig. 17, shows an example of how an agent (blue smiley) should follow the generated path. Also, this operation helps to obtain faster the visible edges from any point, as will be shown in subsequent sections.

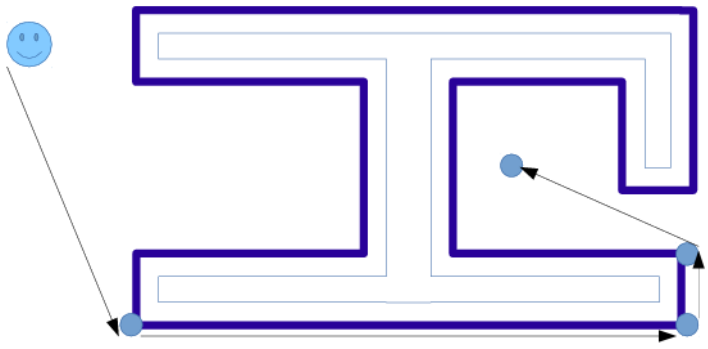

Figure 17: Expanded obstacle polygon

Using SweetHome3D rooms as search areas

One of the most basic parts of the building designing with SweetHome3D is room creation. The program provides a simple way for naming areas and coloring the floor. MASSIS takes advantage of this way that SweetHome3D has for representing the building for speeding up the algorithm. Assuming simple statements, such as:

1. The agent is always inside a room.
2. The target (goal) is always inside a room.
3. A path from Room A to Room B (being A and B different) must cross at least a door.

Although these assumptions seem quite obvious, they make a great improvement in the execution speed of the algorithm. As the obstacle polygons are inflated, their accessible vertices from the inside lie now in the interior of the room shape.
The most expensive operation in terms of computation time is the finding of the visible nodes (vertices) from the start and the goal of the path, because the rest of the visible connections between vertices is precalculated in advance, and the path finding between two nodes in a precomputed graph is done very quickly.
For increasing the algorithm execution speed, the only nodes to be taken into account as starting / ending nodes are the ones inside the room boundaries containing the start point and the end point. In this way, most of the nodes and line segments are discarded, boosting the algorithm speed.

3.4 Elements Localization

There's nowhere you can be that isn't where you're meant to be

John Lennon.

People, sensors and actuators need real time information about the elements that surround them. This implies that, during simulation, lots of query ranges must be performed. It is obvious that iterating over every element on a floor checking distances and intersections is not a good idea. So, there is a need for a data structure capable of holding the simulation elements, being somewhat efficient in data insertion, and retrieval.

An intuitive way for storing the locations of the elements involved during the simulation is using uniform grids. However, these types of data structures are useful when the spatial data is distributed in an homogeneous way, which is rarely seen in MASSIS' simulations. Also, depending on the required accuracy, they can consume too many resources.

Using uniform grids with an appropriate cell size can be a good approach to the problem. But determining the cell size is not an easy task, and it depends on many factors. If they are small, there will be only a few elements per cell but many cells to check. The opposite case is analogous. If the cell size is large, few cells must be checked but there would be many elements per cell.

Another solution could be using QuadTrees [37]. QuadTrees are variable resolution data structures that could be used to retrieve the agent's neighbors within a radius in an efficient manner. This data structure is useful for querying ranges, but it does not perform as well as a uniform grid when inserting and deleting elements. Figure 18 shows an example of space partition using a QuadTree (Only half of it is shown).

Given these facts, the question of which of the two data structures is best for MASSIS arises. Both of them have their respective advantages and disadvantages. Uniform grids are created once,

their elements are contiguous in memory, and insertions and removals are performed almost instantly, but they don't behave as well as a QuadTree for querying ranges. But QuadTrees are costly to construct, and they don't perform as well as uniform grids regarding the insertion and removal of elements, e.g. a moving agent.

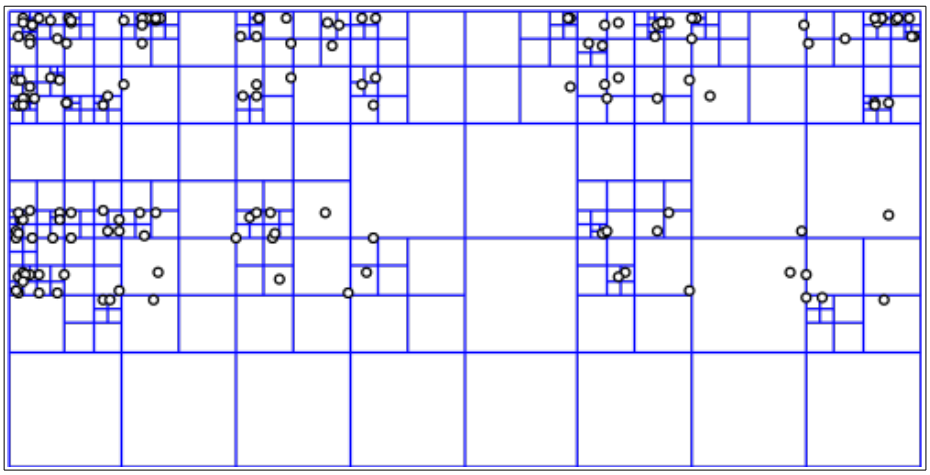

Figure 18: Point QuadTree

The approach taken in MASSIS for solving this issue is simple: If grids perform well at insertion and removal, and QuadTrees at range querying, an hybrid data structure that unifies grids and QuadTrees would obtain both advantages.

3.4.1 Hybrid data structure: : A PR QuadTree

For the reasons mentioned above, a data structure that unifies these two concepts has been designed, with the objective of obtaining both advantages:
constant insertion and efficient range queries. This hybrid data structure consists basically in a quadtree built on top on a uniform grid, which cells are the leafs of the QuadTree. This QuadTree is built only once, with a fixed number of levels.

The approach taken to store the elements in the grid is similar to a PR Bucket QT, which allows each leaf node to store more than one data object, making the leaf a bucket. (The insertion of a data point very close to another in the quadtree might imply many levels

of partitioning in order to have them separated). As the leaves of a are the grid cells, they behave as bucket containers.

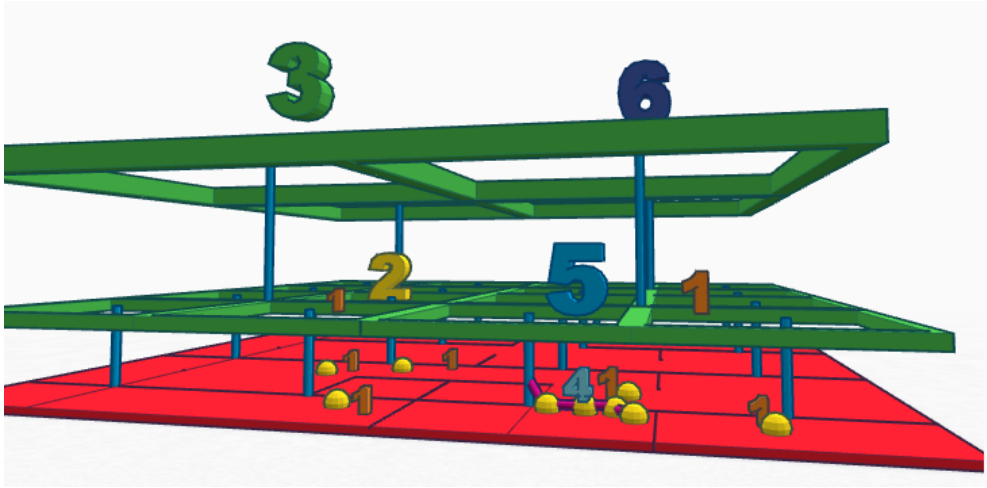

Figure 19: QuadTree 3D representation

Every node in the tree has information of how many data points are under it. When an insertion is done, the leaf node "propagates" the insertion event to its parent, and this to its parent again, until the signal reaches the root of the tree. Figures 19 and 21 show a 3D and 2D graphical representations of this data structure, respectively. Figure 20 Shows a MASSIS' 2D view showing the occupied leaf nodes.

Figure 20: QuadTree layer

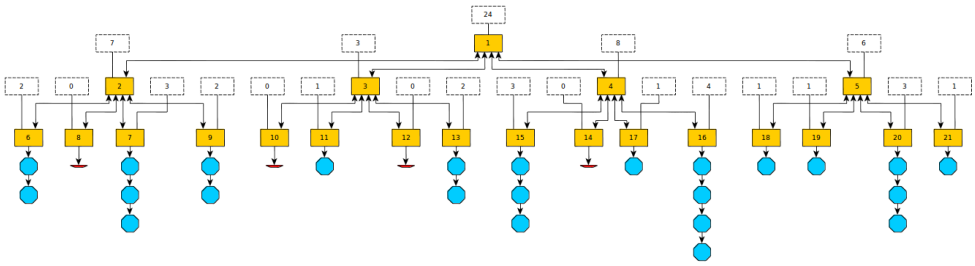

Figure 21: 2D representation as a linked Tree

The most simple way for implementing this type of data structure is using nodes and pointers, to every child and every parent. The insertion is done in the grid, and the insertion event propagation is done accessing the parent of the node until it reaches the root of the tree. Figure 21 shows an example of this data structure implemented as a "linked tree". However, this implementation can be improved, because there is no need for having several parent nodes as objects in memory; they behave as counters of the number of elements under them. Instead, the set of parent nodes can be implemented as a hierarchical array: The size of the array of every level is four times larger than its parent (each node has 4 children).

3.4.2 Bottom-Up Propagation QuadTree Implementation

The implementation of the informally described is implemented in MASSIS as a hierarchical 2D array, where each array in row *n* has four times the size of the array located in the row *n-1*. The parent nodes of the tree are represented by this integer array. However, the implementation of the leaf nodes requires a different array, containing the objects inserted in the tree.

Parent nodes

As we are creating the tree in advance, the nodes marked as "parent" don't need to be actual objects. Instead, the tree structure is created as a hierarchical 2D array, holding only the number of elements under it (as seen in Fig. 21). Figure 22 shows how this data structure is implemented as a hierarchical 2D array.

The arrangement of this array is done following the order NW → NE → SW → SE because getting the parent node index is done

very quickly :
Dividing by 4 the child index has as result the parent index.

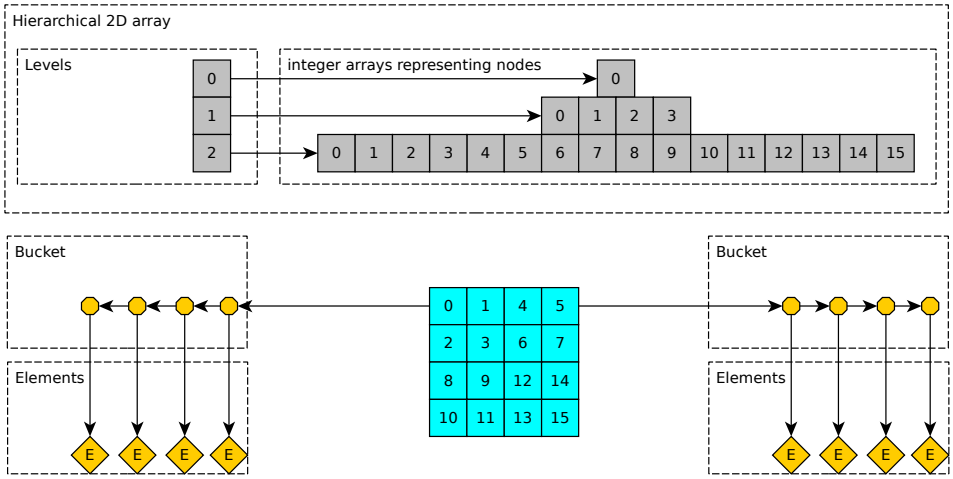

Figure 22: representation as a hierarchical 2D array

As 4 is a power of two, the same result can be archived shifting the bits of the child index two the left twice. (**>>** operator in Java).

Child nodes

The child nodes are a 2D matrix, containing the elements of the bucket in a linked-list arrangement style. Also, they contain the index of the last level of the parent tree. Every time an item is inserted in the grid, the value of the corresponding index of the parent array is incremented by 1.

Element – List node map

In order to obtain constant insertion time, every node must have information about its position on the bucket-list. This can be done in two ways:

1. The element has the node of the bucket list as attribute
2. The contains a O(1) data structure (such as a HashMap) which maps the element with its corresponding bucket list node.

Although the first approach is faster, (it does not need to perform a lookup), breaks modularity; an element can't be stored in

two quadtrees. That's the main reason for choosing the second approach.

Inserting

The insertion algorithm runs as follows:

1. The object coordinates are adjusted to the to the grid coordinates.
2. Get the bucket node associated with this object in the element map. If it does not exist, create a new node for this element and insert it to the map.
3. Propagate *-1* to all the parent nodes, in order to notify that this element no longer is located at the specified cell.
4. Insert the bucket node in the corresponding bucket of the new coordinates.
5. Propagate +1 to all the parent nodes, in order to notify that a new element was inserted.

Range Querying

The range querying is very similar to the standard range query in PR QuadTrees. A recursive search is done over the rectangles intersecting the query range. But there is no need to continue with recursion until the leaf nodes have been reached. The parent nodes contain the number of elements under them; if the counter is zero, the recursive search stops in that branch. The range searches are done via *callbacks*: Small objects containing a *query()* method, which is called every time an element in the given range is found.

Exploring the tree in this ways avoids the creation of new lists. These callbacks should contain also a *shouldStop()* method, which indicates if the search should continue or not. Some types of range queries could be checking the number of elements containing a property, or even if only one agent is in range. That is, The *shouldStop()* method helps for speeding up some queries that only need to check specific elements in the range.

3.5 Steering Behaviors

Don't overcomplicate things: Let your agents move forward towards their goals, changing their course only when it's needed.

Pilar R.Hijas

When the goals of the agent have been determined and the path to the target is known, the agent can start moving in the environment, avoiding collisions with obstacles (e.g. other agents, walls, etc.). These basic skills of the agent can be easily described with *Steering behaviors* [38].

Following the approach taken in [38], the model in which the steering behaviors are implemented in MASSIS is based on a point mass approximation. Every agent with movement ability has as parameters the agent's mass m, the agent's location $\vec{L}$, a maximum force f_{max} and a maximum speed s_{max}.

Every step n, the computed steering forces are applied to the agent's location (limited by f_{max}), producing an acceleration $\vec{A}_n$ which magnitude is inversely proportional to the vehicle's mass.

$$\vec{A}_n = \left(\frac{trunc(\vec{F}_n, f_{max})}{m}\right)$$

The velocity of the agent in every step n is approximated by the Euler integration. Summing the velocity at the previous step ($\vec{V}_{n-1}$) to the current acceleration($\vec{A}_n$), produces a new velocity:

$$\vec{V}_n = trunc(\vec{V}_{n-1} + \vec{A}_n, s_{max})$$

Finally, the velocity is added to the agent's location.

$$\vec{L}_n = (\vec{L}_{n-1} + \vec{V}_n)$$

Fig 23 illustrates these definitions with some examples (Seek and Flee, obstacle avoidance and Path Following).

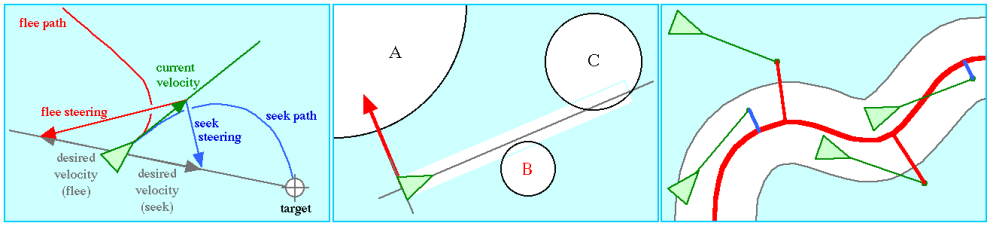

Figure 23: Some steering behaviors:
Seek and Flee, obstacle avoidance and Path Following

MASSIS provides a flexible implementation of several behaviors of this type(e.g. seek, arrival, separation, collision avoidance, wall containment, and path following), that can be grouped into more complex behaviors (like flocking or queuing,for example). Although there are several steering behaviors implemented in MASSIS, for illustration purposes, one variant of the collision avoidance between agents used in MASSIS will be described.

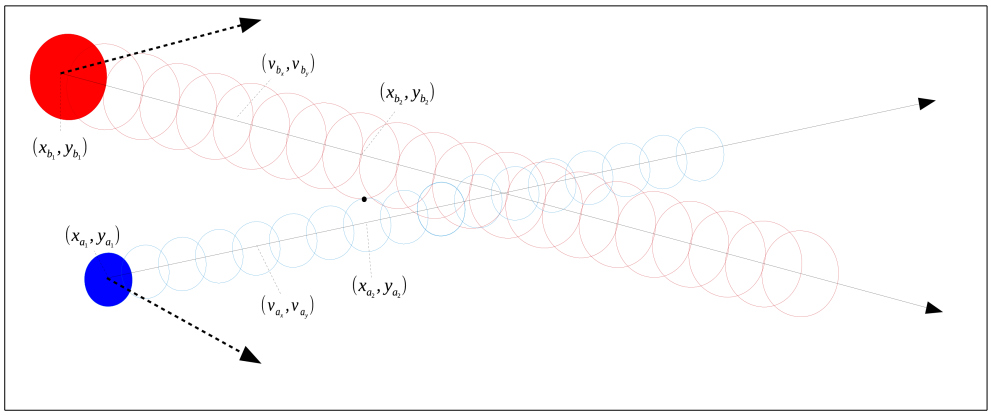

Figure 24: Collision avoidance between two moving agents

In this example, the initial positions of two agents, a and b are defined by their x,y coordinates; (x_{a_1}, y_{a_1}) and (x_{b_1}, y_{b_1}) . If there is a collision, the point where each agent collides with the other one is defined by (x_{a_2}, y_{a_2}) and (x_{b_2}, y_{b_2}) (See Fig. 24). There exists a relation between them, defined by:

$$\begin{aligned} \mathrm{x}_{a_2} &= x_{a_1} + v_{a_x} t \\ \mathrm{y}_{a_2} &= y_{a_1} + v_{a_y} t \\ \mathrm{x}_{b_2} &= x_{b_1} + v_{b_y} t \\ \mathrm{y}_{b_2} &= y_{b_1} + v_{b_y} t \end{aligned}$$

Where t is the time that must pass before the collision occurs. The distance between them should be greater than the sum of their radius.

$$\mathrm{d} \geq r_a + r_b$$

In the time t, the distance between the two elements will be

$$\mathrm{d} = \sqrt{((x_{a_1} + v_{x_a} t) - (x_{b_1} + v_{x_b} t1))^2 + ((y_{a_1} + v_{y_a} t) - (y_{b_1} + v_{y_b} t))^2}$$

Solving for t, in the above equations, two values are obtained: t_1 and t_2 (t_2 is analogous to t_1):

$$\mathrm{t1} = (\text{-}\mathrm{v}_{x_a} x_{a_1} + v_{x_b} x_{a_1} + v_{x_a} x_{b_1} - v_{x_b} x_{b_1} - v_{y_a} y_{a_1} + v_{y_b} y_{a_1} + v_{y_a} y_{b_1} - v_{y_b} y_{b_1} + 1/2(4(v_{x_a}(x_{a_1} - x_{b_1}) + v_{x_b}(-x_{a_1} + x_{b_1}) + (v_{y_a} - v_{y_b})(y_{a_1} - y_{b_1}))^2 - 4(v_{x_a}^2 - 2v_{x_a} v_{x_b} + v_{x_b}^2 + (v_{y_a} - v_{y_b})^2) * (-d^2 + x_{a_1}^2 - 2x_{a_1} x_{b_1} + x_{b_1}^2 + y_{a_1}^2 - 2y_{a_1} y_{b_1} + y_{b_1}^2))^{1/2})/(v_{x_a}^2 - 2v_{x_a} v_{x_b} + v_{x_b}^2 + (v_{y_a} - v_{y_b})^2)$$

The lowest value of t should be chosen, in order to obtain the nearest point of the future collision. An special case must be considered: if the value of t_1 or t_2 is negative, must be discarded (The agents are always moving *forward*). This is illustrated in Figure 25. The pseudo-code of this algorithm is shown in Listing 2.

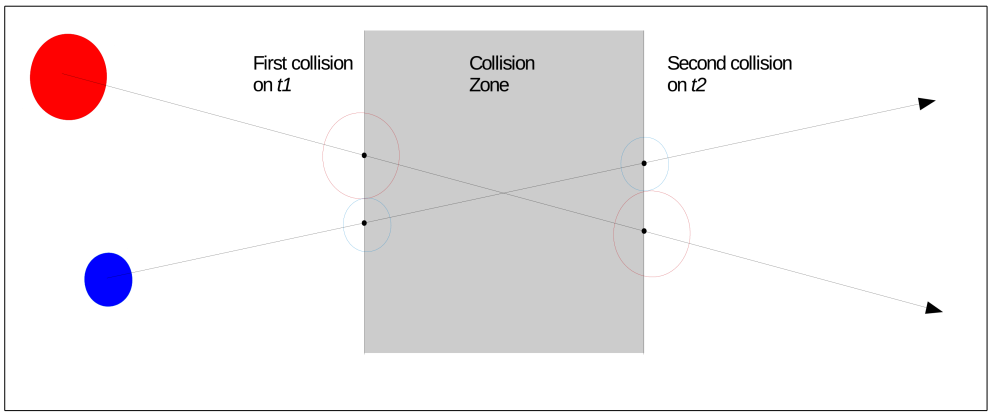

Figure 25: Region division done by the two solutions of t

```
function collisionAvoidance returns repulsionVector
       minT = Infinity;
       for Agent other in getAgentsInRange(agent)
              distance = distance(other,agent)
              d = agent.radius + other.radius
              t = getTimeToCollision(agent, other, d);
              if t > 0 and t < minT
                     minT = t
                     futureA = futureLocation(agent, t)
                     futureO = futureLocation(other, t)
                     repulsionVector = normalize(futureA - futureO)
```

Listing 2: MASSIS' Collision Avoidance pseudo code

3.6 Agent Decision Model

Desires dictate our priorities, priorities shape our choices, and choices determine our actions.

Dallin H. Oaks

The aspects of human behavior that are of interest for modeling indoor scenarios in the MASSIS framework are the mechanisms that humans use to deal with problems reasoning from context, making use of collective intelligence, and how this intelligence is used in problem solving.

For that reason, each agent in MASSIS must have its own behavior, computed by some high-level decision model and executed by a low low level module (speed, position, angles, etc.).

Modeling human behavior with agents have to consider two main aspects: those related with their perception and the interaction with the environment, and those dealing with the reasoning on the context and the decision making. They are implemented as low-level and high-level behavior components, respectively. When the high level component decides what to do next, the action is executed by the low-level component, which performs all the necessary operations. The relationship between these components is illustrated in Fig. 26.

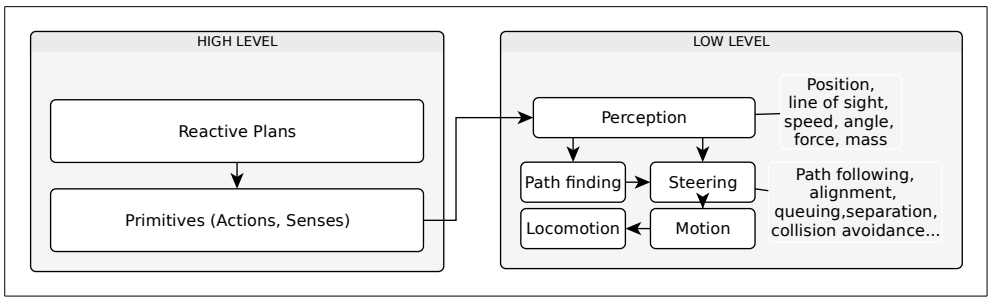

Figure 26: Relationship between high-level and low-level modules

Both high-level and low-level behaviors should be affected by the state of the agent, altering the decision making process (e.g. a scared agent may choose a different route to reach a target, probably a longer one).

During the development of MASSIS, several ways for implementing agent's high-level behavior were considered, and some prototypes were made in order to test which of the different reasoning methods would fit better in this project. It is obvious that there is no clear winner in this area, so MASSIS infrastructure was designed in order to allow an easy integration of different types of AI modules.

Due to time constraints, only one AI module has been fully implemented, but various prototypes based on different techniques were developed, such as Goal Oriented Action Planning. This last one deserves to be explained; if more time and resources were available, it would have been released with the MASSIS final version.

3.6.1 MASSIS' GOAP Behavior Model prototype

GOAP (Goal Oriented Action Planning)[39], [40] proposes a way for action planning in real time for NPCs (Non-Player Characters) in videogames. It was used firstly in F.E.A.R[41], an AAA first person shooter released for PC in 2005.

The planning system it's based partially on the STRIPS Planning [42], consisting on *goals* and *actions*. Goals describe some desired state of the world, and actions are defined in terms of preconditions and effects.

The actions only can be executed if all its preconditions have been met, and every action changes the world in one way of another.

The way for linking preconditions, actions and effects is done using A*, where the world states are the nodes of the graph and edges are the actions to reach them.

The MASSIS GOAP Prototype was based in this idea. The main concepts are the following:

1. **(Pre)Condition:** Circumstance necessary and indispensable for an action to be carried out.
2. **Action:** "Do something". It requires certain preconditions to be executed, and causes effects.
3. **Effect:** Impact or consequence of an action. It Adds or changes the *real* or *perceived* world.
4. **World:** The building.
5. **Perception (Sense):** Information managed by the agent, in relation with its environment. There are two types of Senses: *Real / External* senses and *Internal* senses.
    - *Real / External* senses: They are the real world facts perceived from the agent. For example:
        - "There are **5 items** on that table"
        - "There is **a fire** on this room"
        - "Somebody **is looking at me**"
    - *Internal* senses: They are the things perceived from the *internal world* of the agent. Internal senses are the result of processing the External senses into something that exists only in the agent's mind. For example:
        - "There are a **lot of items** on that table"
        - "I am **not comfortable** in this room"
        - "That person **makes me nervous**"

The preconditions are tightly coupled with perceptions, being in most cases the same.

Action-Condition-Effect blocks

Different *actions* can provoke the same effect(s), but different *preconditions*.

For example, if the goal is to have a comfortable chair, different actions can achieve this goal:

Action: Buy a chair (we omit the going-to-Ikea part)
    Condition: Have money.
    Effects:
        Have a comfortable chair.
        Have less money

Action: Buy tools for making chairs.

Condition: Have money.
Effect:
Have tools for making chairs.
Have less money.

Action: Borrow tools for making chairs from a friend.
Condition: Have friends with tools of this type.
Effect:
Have tools for making chairs.

And much more of these *Action-Condition-Effect* blocks should be necessary for giving the agent the ability of making a decent plan.

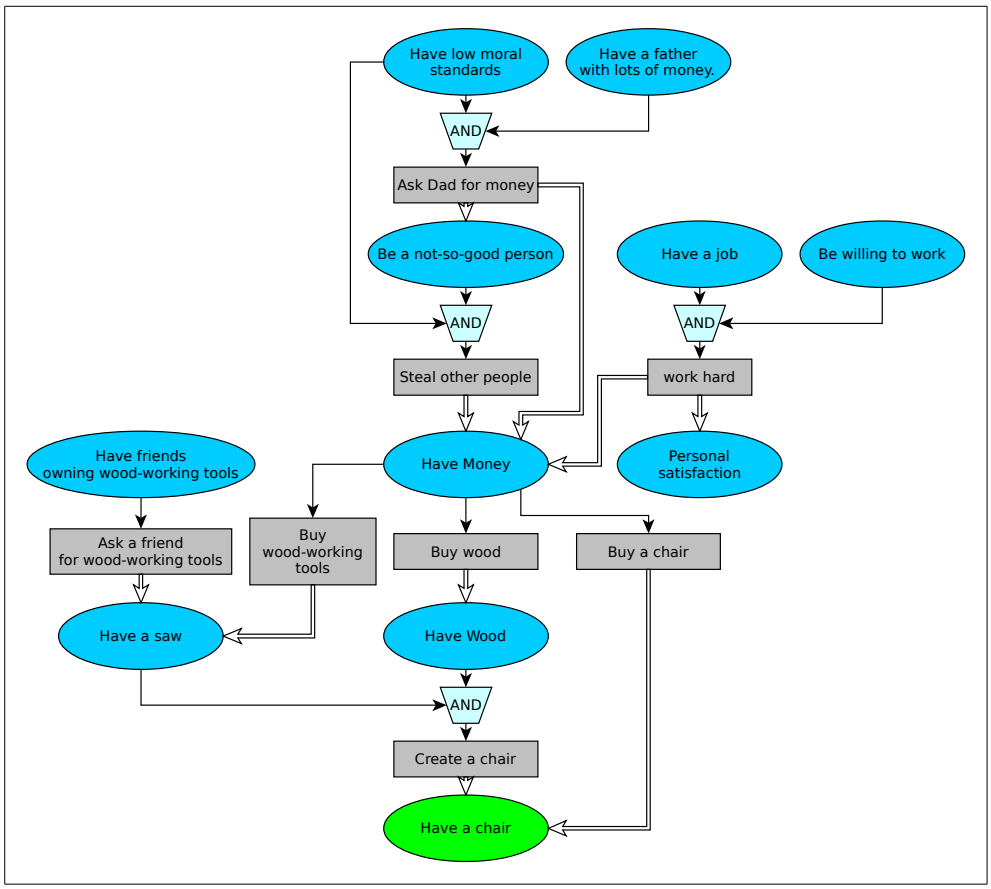

Figure 27: Joined Action-Condition-Effect blocks

Depending on the agent's perception of the world, this blocks are linked together dynamically, and a graph search is performed on the

Action-Condition-Effect graph. Figure 27 shows a simple example of joining the preconditions with the actions and effects.

Behavior

In this context, a behavior is a set of actions that must be performed (ordered or not) in order to reach a goal. Basically, it consists on some conditions, which "fire" its activation and other conditions which invalidates them.

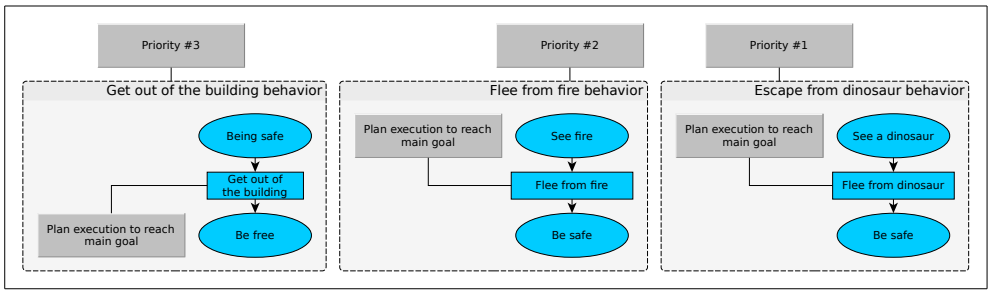

Figure 28: Behavior priorities in MASSIS' GOAP Variant

Behaviors are ordered by priorities: Some events in the world change the way we behave: For most people, seeing a dinosaur coming changes their short-term priorities. An example of behavior priorities is illustrated in Figure 28.

Behavior stack

When a behavior becomes invalidated by some event, the agent must continue doing whatever it was doing. That is the reason why events in the environment do not replace the current behavior, they add it to a behavior stack.

The agent always has on the bottom of its behavior stack the lowest- priority behavior. When an event forces the agent to change it, a new behavior is added on the top of the stack. If a firing condition of a lower level behavior is satisfied, that behavior is not added to the stack; it has a lower priority and shouldn't be taken into account. Figure 29 illustrates an example of the management of a behavior stack depending in the environment.

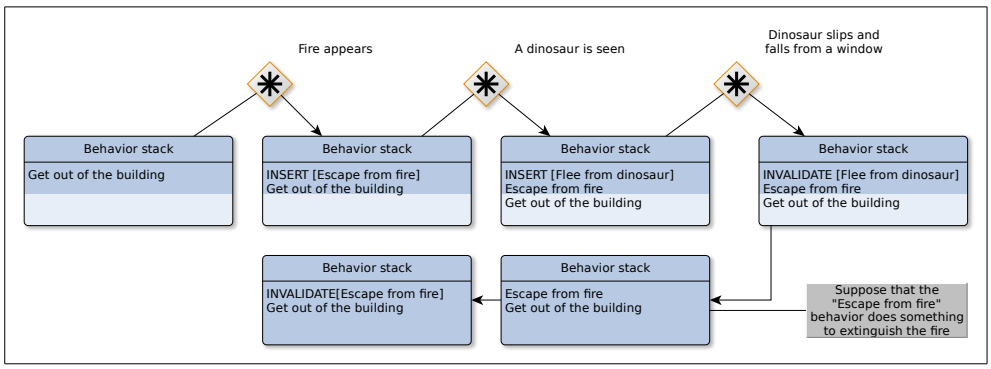

Figure 29: Behavior stack changes due to events in the environment

Case study for this decision model : Killer – Prey

For testing this decision model, a Killer – Prey scenario was modeled. In this scenario, three roles were present:

1. **Killer :** Wants to kill every agent marked as Prey, but for killing a prey a Knife is needed. If it does not see any prey agent, wanders around.
2. **Prey:** Wanders around until it sees a Killer. When a Killer its seen by a Prey, this last one flees as fast as possible.
3. **Knife:** A simple knife. Can be taken by a Killer, and they are distributed through the building. For making the simulation more interesting, a knife can be used only once.

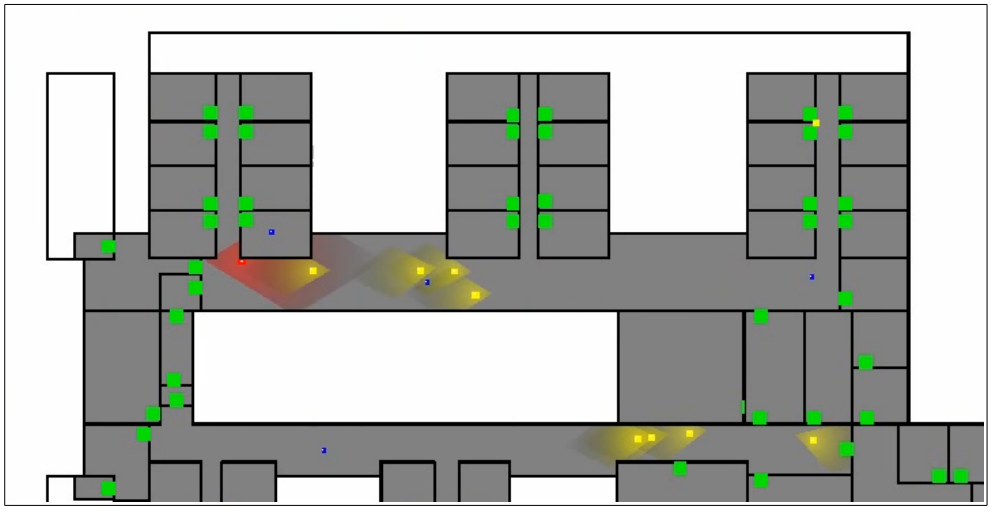

Figure 30: Killer – Prey Prototype screenshot

Listing 3 shows part of the trace of the simulation, and Figure 30 shows an screenshot of it. The red agent is the killer, the yellow ones the preys, and the blue dots are knifes.

```
============Begin Planning============
Initial state : {isPreyKilled=0.0, NumberOfKilledPeople=0.0, NumberOfKnifesVisible=0.0,
isPreyVisible=0.0, NumberOfPreysVisible=0.0, isKnifeVisible=0.0, isAlive=1.0, isPerson=1.0}
Available Actions : [TakeKnife, WanderAround, KillPreyAction, SearchKnife, SearchPrey]
Planning for goal [WanderFinished:==1.0]
--------------------------------------
Actions : [WanderAround]
============End Planning============
[STATE]Executing Action
The perceived environment has changed, forcing a change on the agent behaviour.
Firing condition: [isPreyVisible:==1.0] /* A prey was seen */
Current behaviour: Behaviour [Kill Prey Behaviour]
Behaviour stack: [Behaviour [Wander around], Behaviour [Kill Prey Behaviour]] /* Behavior added to
the stack */
[STATE]idle - Replanning next goal
============Begin Planning============
Initial state : {isPreyKilled=0.0, NumberOfKilledPeople=0.0, NumberOfKnifesVisible=0.0,
isPreyVisible=1.0, NumberOfPreysVisible=1.0, isKnifeVisible=0.0, isAlive=1.0, isPerson=1.0}
Available Actions : [TakeKnife, WanderAround, KillPreyAction, SearchKnife, SearchPrey]
Planning for goal [isPreyKilled:==1.0]
--------------------------------------
Actions : [SearchKnife, TakeKnife, SearchPrey, KillPreyAction]
============End Planning============
[STATE]Executing Action
Action SearchKnife Finished.
WorldState : {isPreyKilled=0.0, NumberOfKilledPeople=0.0, NumberOfKnifesVisible=1.0,
isPreyVisible=1.0, NumberOfPreysVisible=3.0, isKnifeVisible=1.0, isAlive=1.0, isPerson=1.0}
[STATE]Executing Action
Action SearchKnife Finished
[STATE]Executing Action
[STATE]Executing Action
Action TakeKnife Finished
Action SearchPrey Finished.
WorldState : {isPreyKilled=0.0, NumberOfKilledPeople=0.0, HasKnife=1.0, NumberOfKnifesVisible=1.0,
isPreyVisible=1.0, NumberOfPreysVisible=1.0, isKnifeVisible=1.0, isAlive=1.0, isPerson=1.0}
[STATE]Executing Action
Action SearchPrey Finished
[STATE]Executing Action /* Prey was stabbed */
The perceived environment has changed making the actual behaviour(Behaviour [Kill Prey Behaviour])
invalid.  /*Prey is now dead and there is not any other visible*/
[STATE]idle - Replanning next goal
```

Listing 3: MASSIS' GOAP Case study trace

3.6.2 MASSIS' behavior model: POSH Plans

While the method described above looks promising, the development and testing of it would have taken too much time. Therefore, it was decided to use a more mature behavior model, well known and tested: POSH[43] (Parallel-rooted, Ordered Slip-stack Hierarchical) dynamic plans.

The architecture of this behavior follows the BOD (Behavior Oriented Design) method[43], [44]. This method for building agents combines the advantages of Behavior Based AI[45], [46] and object-oriented design approaches.

In MASSIS this is applied to facilitate the design of agents that are capable of running in parallel and of generating a behavior that can satisfy multiple objectives that may conflict with each other.

One of the main issues while designing an autonomous agent is that many of the goals that the agent wants to be accomplished

must be carried out at the same time. An agent may have the desire to be loved, be promoted at work and having breakfast in the morning, and talking with a colleague about some book. Additionally, these goals must be achieved in an unpredictable environment,which can complicate or even make easier the way in which the agent tries to accomplish its goals.

For example: If the agent's *original* plan for accomplishing its goals is the action sequence $A \rightarrow B \rightarrow C \rightarrow D$, is possible, due to a change in the environment, that the action B cannot be executed, and a more complex way must be chosen, or the opposite; the change in the environment triggers the availability of the action E, which makes easier the goal accomplishment.

Developing a system of agents under BOD involves dividing the implementation into two different parts: A library of behavior modules and a set of POSH Plans.

A library of Behavior modules.

They consist of a set of classes representing a set of modules for perception, action and learning. These are *primitives*; actions and senses that can be called from the mechanism of action selection.

They also provide a place where certain states and knowledge can be stored in order to perform those actions, and they contain code that describes any sense that needs to be carried out to acquire that state and knowledge. In brief, they determine *how to do something*. These senses and actions are created in the native language for the problem space, in the case of MASSIS, Java. Thanks to Pogamut's implementation, the primitives can be defined in an easy and modular way. They can be expressed as a set of methods grouped in behaviors (Listings 4 and 5) , annotated with the interfaces `@SPOSHSense` and `@SPOSHAction`, or as a set of classes, each one representing a primitive (Listing 6).

```
@SPOSHSense
public boolean stuck() {
	boolean s = this.getAgent().isStuck();
	System.out.println("stuck() = " + s);
	return s;
}
```

Listing 4: Example of a primitive sense definition in Java as a simple method

```
@SPOSHAction
public ActionResult move() {
	boolean move = this.getAgent().move();
	System.out.println("move() = " + move);
	return ActionResult.FINISHED;
}
```

Listing 5: Example of a primitive action definition in Java as a simple method

```
@PrimitiveInfo(name = "Sees Element",description =
   "True if sees another with the attribute and value provided")
public class SeesElement extends SimulationSense{
    public SeesElement(SimulationContext ctx) {
        super(ctx);
    }

    public Boolean query(
                        @Param("$attr") String attr,
                        @Param("$value")Integer val) {

        for (Agent v : this.getAgent().getAgentsInVisionRadio())
        {
             if (v.getProperty(attr)==val) {
                   return true;
             }
        }
        return false;
    }
}
```

Listing 6: Example of a primitive sense definition in Java as a class

Pogamut's extension allows the parametrization of the plan (and, subsequently, the plan's primitives). This feature boost flexibility and eases the code reuse. As shown on Listing 6, depending the the parameters *$attr* and *$value*, the result of the sense query would vary.

POSH Dynamic action selection scripts.

POSH (Parallel-Rooted, Ordered Slip-Stack Hierarchical) Dynamic action selection scripts allow to determine priorities between modules, and BOD architecture uses a POSH dynamic plan when an action should be carried out.

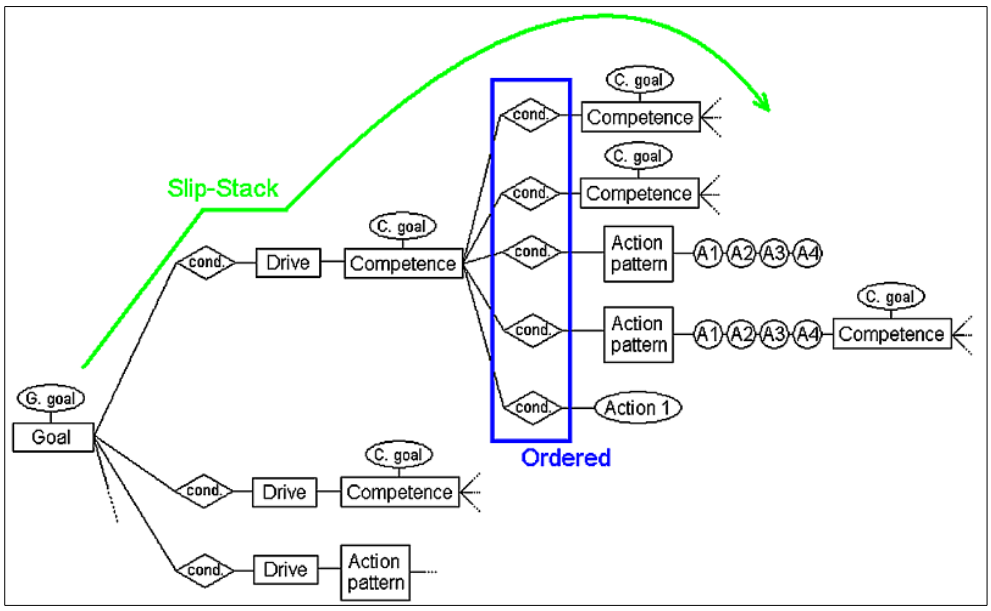

Figure 31: Transversal of a POSH plan

A POSH plan is a prioritized set of conditions and the related actions to be performed when the conditions have been met. It consists of drive collections, competences, and action patterns. Figure 31 shows how a POSH plan is transversed.

Drive collections

Drive collections are the root of every POSH plan. On the action selection step, the drive collections select which goal the agent must try to to accomplish. They can be seen as a set of conditional rules, that are evaluated from highest to lowest priority.

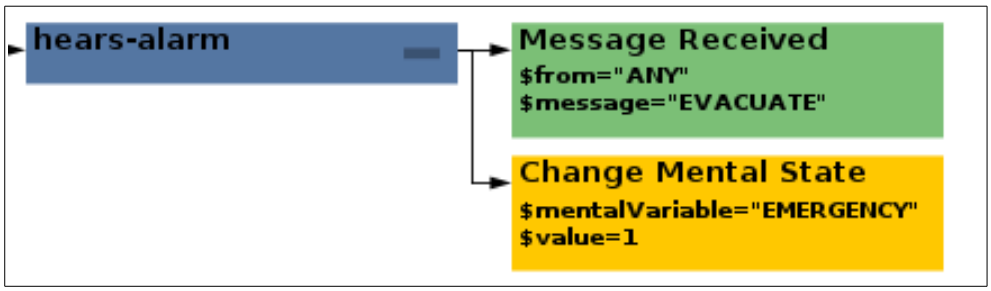

Figure 32: Mental state modification in a POSH plan (yellow), triggered by the sense
"Message Received" (green) on the Drive Element "hears-alarm"(blue)

Every time the condition of the drive collection with highest priority is satisfied (a higher rule interrupts a lower one), the POSH engine executes the corresponding action pattern or competence. Figure 32 shows a graphical example of a Drive Element (*hears alarm*).

Competences

Competences are a set of nested *if-then* conditional trees, which can be reused several times inside the reactive plan. They differ from the drive collections in the way they are executed; rules they do not interrupt each other.

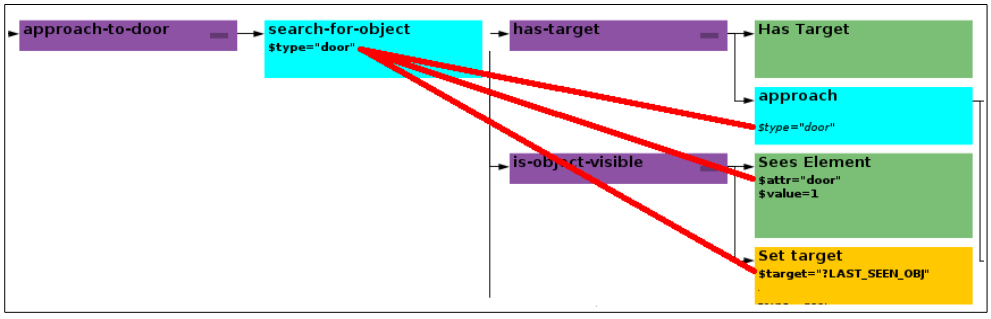

Figure 33: Propagation of the value "door" in the parametrized competence of searching an object.

Action Patterns

Action Patterns are simple sequences of actions. Although they are not very flexible, they provide a layer of abstraction very useful when grouping actions.

Case Study for this decision model : Emergency Simulation

Public buildings have some protocols for dealing with emergency situations, which may involve, for instance, evacuation of the building. Testing these protocols requires some planning and cost. Simulation can help to this task.

This case study addresses this kind of situation for the building of the Facultad de Informática at UCM. In this scenario, a teacher is responsible for guiding students safely to the building's exit in case of emergency. For illustrating purposes, this is the protocol for a teacher in an emergency situation:

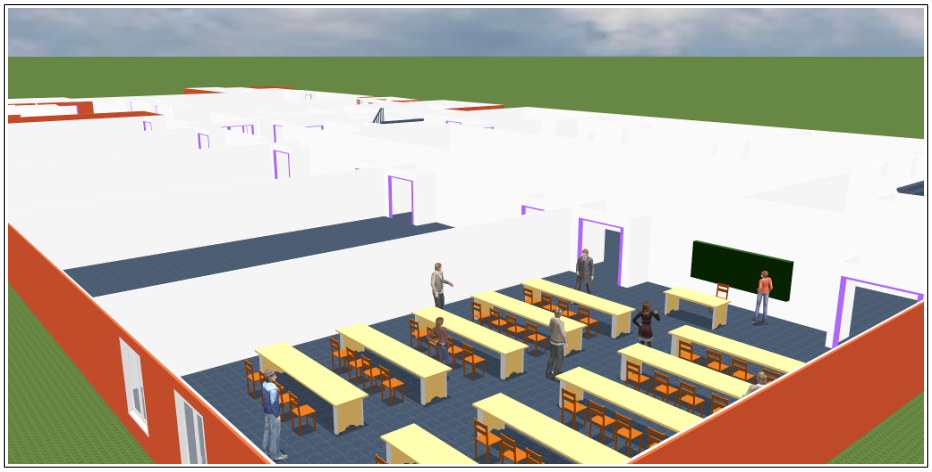

Figure 34: Initial state of the simulation, in a 3D view

When the alarm sounds the teacher of the group should go to the classroom door and order the students to close the windows if there is a fire. If instead of a fire there is a bomb threat, windows and doors should be left open. Students will leave the classroom through the door and they will be waiting for the teacher outside. The teacher will be the last person to leave the classroom. Once there is nobody in the classroom, the teacher will place a chair at the entrance of it,as an indication that the room has been evacuated entirely. Then the teacher will guide students toward the nearest exit.

So, here our teacher agent must have the following basic skills:

- Hearing and vision capabilities.
- It must be able to communicate with other agents by voice.
- Movement.

- Interaction with objects in the environment: Taking an object, carrying it, dropping it.

These skills are candidates to be primitive actions and senses. These primitives, are used by the reactive plan as Triggers of *Drive Elements*, components of *Action Patterns*, or they form part of one or more *Competences*. Figure 35 shows the teacher's reactive plan, and Figures 32 and 33 are some of the POSH plan components used in this example.

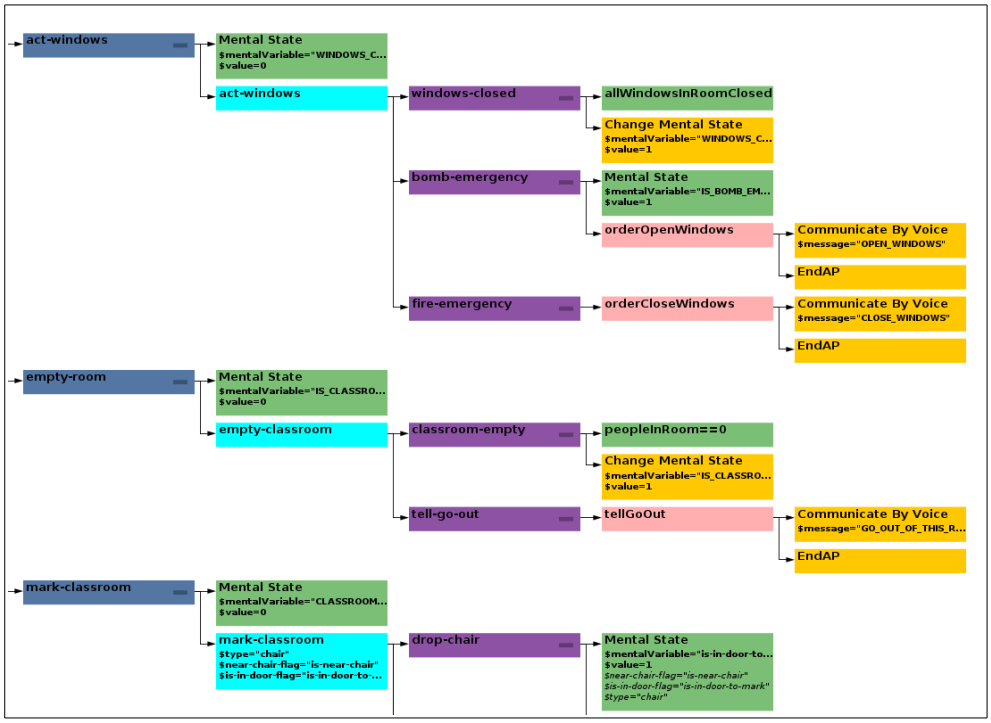

Figure 35: Partial overview of the Case Study POSH plan

Figures 34 and 36 show the initial state of the simulation. When the alarm sounds, and the teacher goes to the door (using *go-to-nearest-door competence*). Figure 37 illustrates the moment when the teacher tells the students that they must close the windows (act-windows competence). When the windows are closed, and the students outside (Fig. 38), the teacher takes the first chair he sees and he moves it to the door. After that, the teacher guides his students to the nearest exit (Fig. 39).

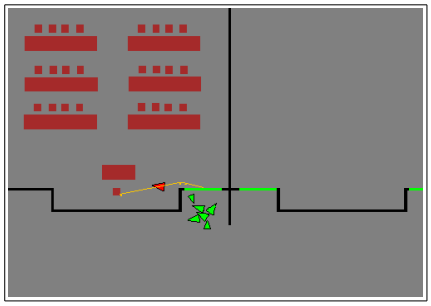

Figure 38: Teacher going to take the nearest chair

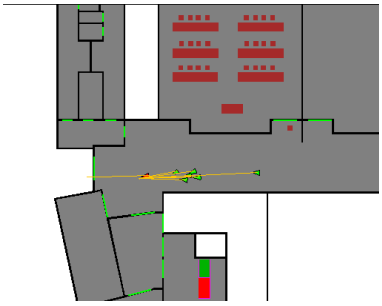

Figure 39: The students follow the teacher for escaping from the building.

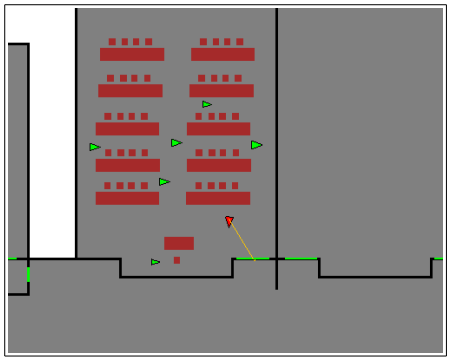

Figure 36: Teacher going to the door

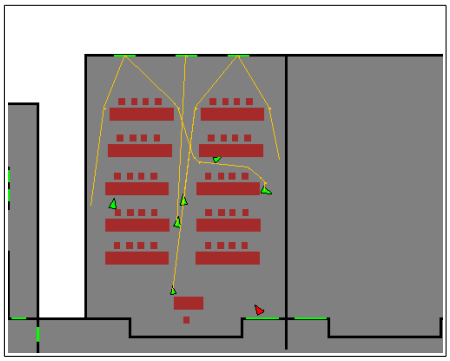

Figure 37: The students proceeding to close the windows.

3.7 Saving & Loading Simulations

I think you can learn from history.

Chuck Norris

Although MASON provides a way for saving simulation states, it does not provide a way for storing some kind of *record* of the entire simulation. That's the reason why a new module for saving and loading the simulation results was developed. Two ways for saving simulations were considered when developing MASSIS: Java Serialization format and a plain-text format.

- **Java serialization**:
  - Good points:
    - Easy implementation
    - If it is done carefully, the disk space used is relatively small.
  - Bad points:
    - It has high dependence from the codebase
    - Makes very difficult analyzing the simulation results from other platform/language.
- **Plain text**:
  - Good points:
    - It can be parsed by any platform and language
    - It is human-readable
  - Bad points:
    - Can use high amounts of disk space
    - A custom parser is needed, and also an specification of the file format.
    - Making changes in the codebase may imply rewriting too much code, if everything is done “by hand”.

Balancing the advantages and disadvantages of these two ways for storing the simulation timeline, the second approach (Plain – Text) was chosen, but using JSON (JavaScript Simple Object Notation)[47] as file format. JSON is a well known text format, suitable for storing object definitions. For logging the simulation state to this format, an extension of the well-known *Gson* [48] library, *Gson on Fire* [49] was used. It serializes the object attributes into JSON format, and also provides several utilities for solving typical issues when doing these things (e.g. circular references).

3.7.1 Saving the simulation

The simulation changes can be saved directly in a (zipped) text file, or in a SQLite [50] database.

Using SQLite

SQLite was chosen because is a very portable database format. The database contains only 3 fields: **<object-id>**,**<step-id>**,**<object-state>**. **<object-id>** it's the simulation object unique id, **<step-id>** the time-step of the simulation and **<object-state>** is the actual state of the simulation object. The main advantage of this way of storing the simulation results is that making a simple select statement, all the changes that the agent made during simulation can be retrieved (See Listing 7).

```
select simulation_object_id,
       max(step_id),
       simulation_object_state
from simulation_log
where step_id < DESIRED_STEP
group by simulation_object_id
```

Listing 7: SQLite query to retrieve the changes made by an element of the simulation.

Although this is not a bad approach, the number of records in the table grows very quickly, and its file size has an extra overhead regarding the *pure-plain-text* approach.

Using Plain Text

Much more simple than the SQLite option: one line per step, containing the elements that changed in that step. Very fast to

write and to playback forward (but not backwards, that option is better with the SQLite approach).

Saving Disk Space

JSON is a compact text format (compared with XML). The file size can be reduced considerably, if some methods are applied.

ZIP compression

Applying ZIP compression to the simulation results reduces notoriously its file size: up to 75%. If the simulation results are saved into a SQLite database, the database must be compressed after, but if they are saved directly into a file, the stream can be compressed "on the fly" (That is, less space consumed overall).

String substitution

Although the ZIP compression achieves good results, some optimizations can be made in order to decrease even more the file size. Listing 8 shows an example of an agent state in a concrete simulation step.

```
{
    "velocity": { "x": 1, "y": 1 },
    "visionRadio": 300,
    "maxforce": 10,
    "maxspeed": 15,
    "properties": {
      "steering.alignment": 5,
      "isPrey": 1,
      "steering.separation": 70,
      "steering.followpath": 25,
       //...
    },
    "locationState": {
      "angle": 0.7853982,
      "floorId": 8,
      "centerX": 4975.2285,
      "centerY": 4108.2695,
      "type": "SimLocationState"
    },
    "id": 3673,
    "type": "AgentClassNameState"
  }
```

Listing 8: Un-compressed agent state in JSON format.

Blue strings are repeated with a very high frequency (they are object attributes, or class names). Attributes like "`velocity`" are common to every movement-capable agent, e.g. a person. If the 8

characters that "`velocity`" has can be substituted by something shorter, the overall size of the simulation results file would be considerably smaller.

```
{
  "@D": { "x": 1, "y": 1 },
  "@E": 300, "@F": 10, "@G": 15,
  "@H": { "@6": 5,"@7": 1,"@8": 70,"@9": 2, "@A": 25,...},
  "@I": {
    "@0": 0.7853982, "@1": 8,  "@2": 4975.2285, "@3": 4108.2695,
    "@5": "@4"
  },
  "id": 3673, "@5": "@J"
}
```

Listing 9: String substitution in an agent's saved state.

The keys (representing object attributes), and frequent string values (such as class names) can be replaced by a shorter string, something like a number preceded by an special character that serves as indicator, like "`@`". But, why a number? Instead of a number can be an alphanumeric character, so, instead of working in base 10, we are working in base 62. Listing 9 shows how the agent state is represented after this "reduction" process.

```
[
      ["@0","angle"],["@1","floorId"],
      ["@2","centerX"],["@3","centerY"],
      …
]
```

Listing 10: Array mapping key alias with actual keys

It is obvious that some kind of extra information must be stored in order to be capable of reverting the process. So, a mapping array must be saved together with the simulation results. An example of this mapping array is shown in Listing 10.

Using concurrency for speeding up the process

I/O operations are costly. As every simulation step implies writing to disk (if saving is enabled), the execution speed of the simulation can be reduced significantly.

MASSIS uses an extra thread for writing this information. A synchronized queue is used as writing buffer. Every step, the infor-

mation about the changes is pushed into the queue. The writing thread is continuously taking elements from the queue and writing that information to disk. In this way way, the impact of I/O operations is lower.

3.7.2 Simulation Playback

A simulation playback is the reverse process mentioned above: The data is loaded from the file, and applied to the simulation objects. The way for doing this is fairly simple: If the simulation is started in *Playback mode*, the proper *step()* method of the agent is not called. Instead, the information retrieved from the simulation file is loaded into the agent, changing its properties directly.

3.8 Visualization

You must see first before you can believe.

Raymond Holliwell

The visualization of what's happening during the simulation is one of the most important parts for any simulation framework. Although MASON provides good support for this, it is more oriented to grid-based models, and adapting the MASON display modules to meet the requirements of MASSIS entailed more work than creating a new display system. MASSIS visualization is composed in two parts: 3D visualization and 2D visualization. The 3D visual interface is more suitable for having a realistic view of the building, and what is happening inside, but the 2D display is more useful for testing and debugging the model. Also, it is easier to the programmer to extend and manage.

3.8.1 3D Visualization

The 3D view is based on SweetHome 3D integrated view. It shows the building and the elements inside it in the same way that SweetHome 3D does, but some modifications must have been made in order to make it faster for displaying the model in real-time.

That is because SweetHome3D is intended for being used as a home-design program, not as a simulation model display. SweetHome 3D it has been designed using a MVC pattern, and has lots of event listeners that are executed when an element of the building is changed.

Figure 40: MASSIS' Real-Time 3D Visualization

Although that's a good way for structuring an application like this, it's not needed for MASSIS. In addition, it is not thread safe. The events listeners were removed, and also some intermediate layers, in order to gain execution speed. MASSIS architecture accesses straightforward to the 3D modules, avoiding intermediate calculations. Figure 40 shows an example of MASSIS' 3D visualization.

3.8.2 2D Visualization

Layer-Based Display

MASSIS' 2D display is based on *layers*. Each layer can be designed independently, and it is intended for displaying the simulation in an schematic view.

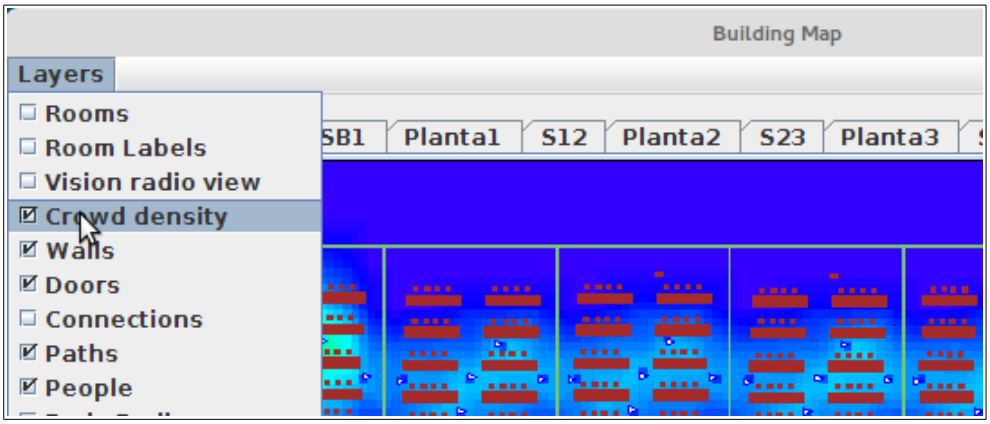

Figure 41: MASSIS' Layer Selector

Layers are drawn one on top of each other, following an order. They can be ordered, enabled and disabled. Listing 11 shows the code needed for one of the most basic layers: Walls. Figure 41

```
public class WallLayer extends FloorMapLayer {
        public WallLayer(boolean enabled) {
                super(enabled);
        }
        @Override
        protected void draw(Floor floor, Graphics2D g) {
                g.setColor(Color.GREEN);
                for (SimWall wall : floor.getWalls())
                        g.fill(wall.getPolygon());
        }
        @Override
        public String getName() { return "Walls"; }
}
```

Listing 11: Example of one of the most basic MASSIS' layers: Wall Layer.

shows how the layers can be enabled or disabled directly from the GUI, from an user-friendly checkbox list.

MASSIS' built-in layers

MASSIS comes with several *basic* built-in layers, for showing rooms, doors, agents, radios, paths, pathfinding utilities...etc. Figure 42 shows a combination between the obstacle layer, agents layer, agent's radius layer, pathfinder layer,walls layer,rooms layer and doors layer.

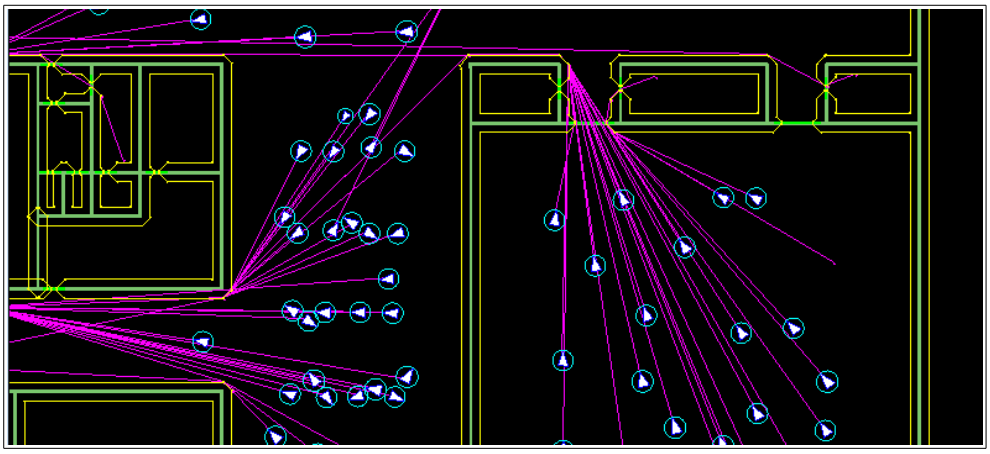

Figure 42: Layer combination example

However, new layers can be designed without modifying the simulation engine.

4 Getting Started With MASSIS

The secret of getting ahead is getting started.
Mark Twain

4.1 Installation

4.1.1 Downloading MASSIS

MASSIS can be downloaded from its main repository, `https://github.com/rpax/MASSIS`. The dependencies of the project have been managed with Maven, making easier the dependency management. For getting MASSIS and all its dependencies:

1. Download maven (`https://maven.apache.org/download.cgi`)
2. Clone MASSIS repository (or download it as a zip file)
3. generate MASSIS executable using Maven:
   ```
   mvn clean install
   ```
4. This should generate a zip file (MASSIS.zip) containing the executable jar file and all its dependencies.

Some of the dependencies of the project aren't in any public maven repository (such as SweetHome3D). To solve this issue, a release tag has been created, containing these required libraries and MASSIS' plugins jar file. The contents of `libs.zip` must be placed under the folder `rpax.tfg`. The url of this tag is

`https://github.com/rpax/MASSIS/releases/tag/v1.0`

The installation process of MASSIS is minimal. In fact, if the environment designer is not needed, it does not require installation at all. For using MASSIS' SweetHome3D plugins, the MASSIS plugin file (`rpax.tfg.sh3d.plugins.metadata.MASSISMetadataPlugin.jar`) must be

placed in the SweetHome3D plugin folder. Depending on the OS, it can be one of the following:

Windows

- Windows Vista, Windows 7 & Windows 8

```
C:\Users\<user>\AppData\Roaming\eTeks\Sweet Home 3D\plugins
```

- Windows XP

```
C:\Documents and Settings\<user>\Application Data\eTeks\Sweet Home 3D\plugins
```

MAC OS X

```
~/Library/Application Support/eTeks/Sweet Home 3D/plugins
```

Linux & Unix

```
~/.eteks/sweethome3d/plugins
```

4.2 Environment Creation

4.2.1 Launching the environment editor

For launching the environment editor, MASSIS must be started with the `EDITOR` argument for the parameter `run-as`.

```
java -jar MASSIS.jar --run-as EDITOR
```

4.2.2 Designing the environment

Creating & Editing walls

The creation of walls can be done in two ways:

- Selecting Plan → Create walls (Figure 43)
- Pressing the  icon.

After that, every left click in the map will create a wall. For finishing the *wall creation mode,* just click the arrow icon or press the `Esc` key. (Figure 44)

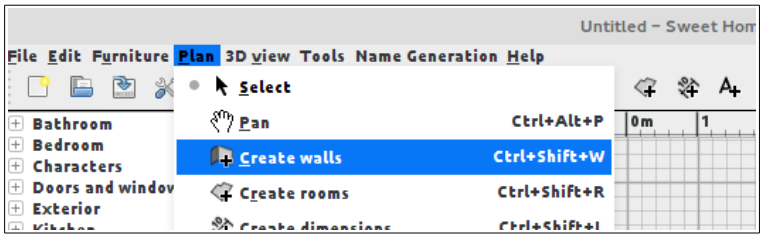

Figure 43: Selecting the Create walls option

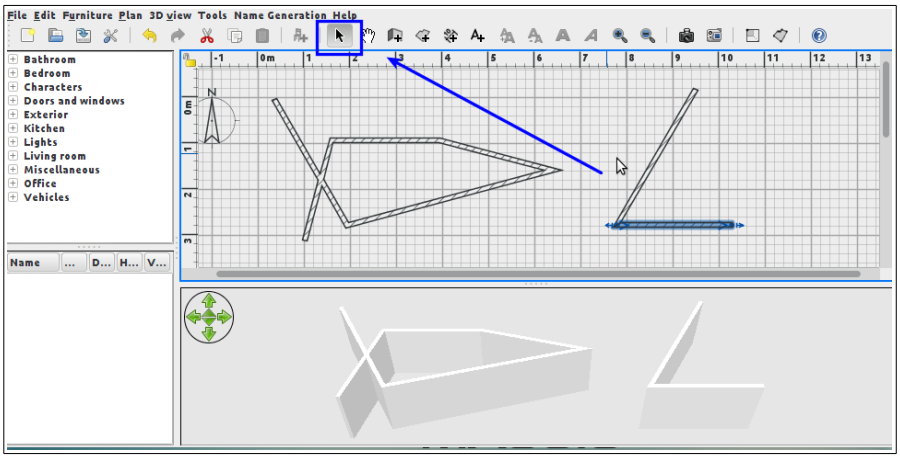

Figure 44: Editing walls

Adding doors, windows & furniture

The furniture, doors and windows can be selected from the left pane, and can be placed into the building by dragging and dropping them. **Note**: it is important to place the doors and windows in the walls. Otherwise, the simulation may not work correctly. Figure 45 illustrates this.

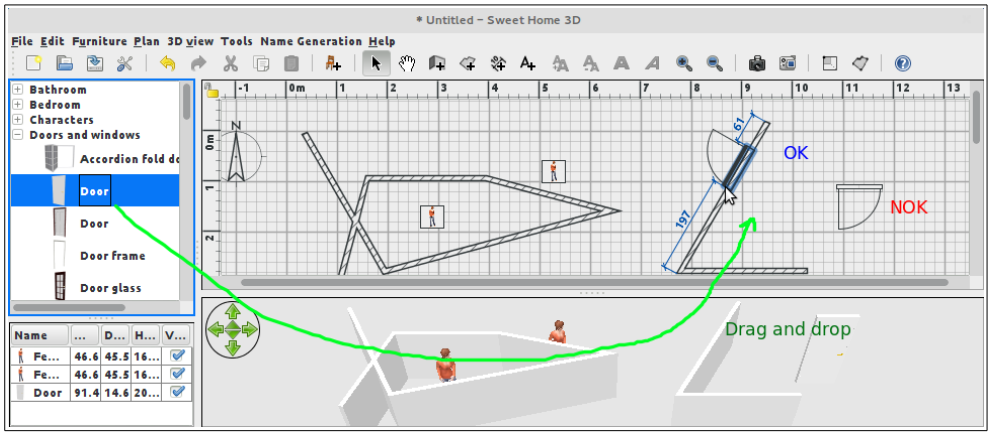

Figure 45: Adding doors, windows and furniture

Importing 3D objects

Extra 3D objects can be imported, and configured as window/door/furniture, from *Furniture* → *Import Furniture*.

Drawing rooms

Rooms can be added the same way as walls, using the  button or selecting the option in *Plan* → *Create Rooms*.

Very important note: This way is highly discouraged for creating environments for MASSIS. As MASSIS uses the rooms in its preprocessing algorithms, a bad placement of them can make the simulation behave incorrectly, or even failing. The preferred way is double clicking a closed area, surrounded by walls. In this way, the new room will occupy the entire space. Figure 46 shows the steps for doing this task.

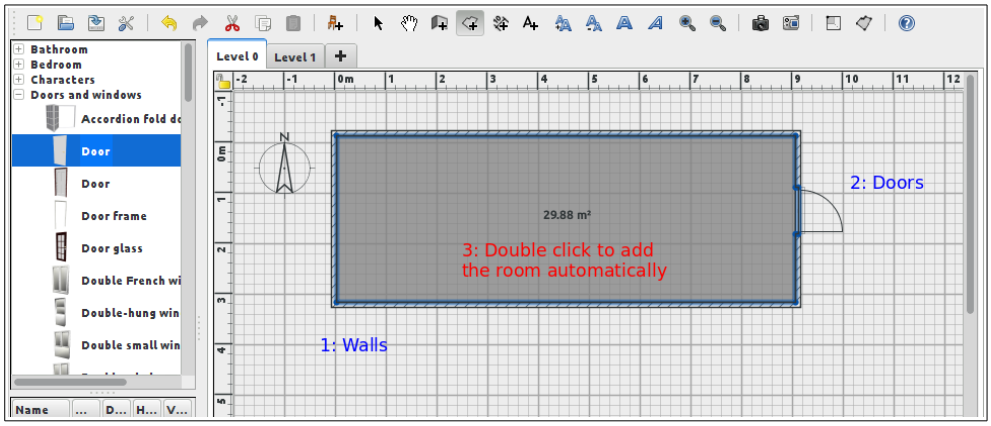

Figure 46: Steps for creating a Room

Adding levels

Levels can be added by clicking the **+** icon in the Levels tab, as it is shown in Figure 47.

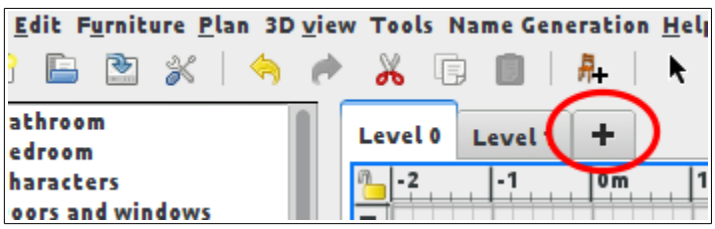

Figure 47: Adding levels

4.2.3 Metadata Editor

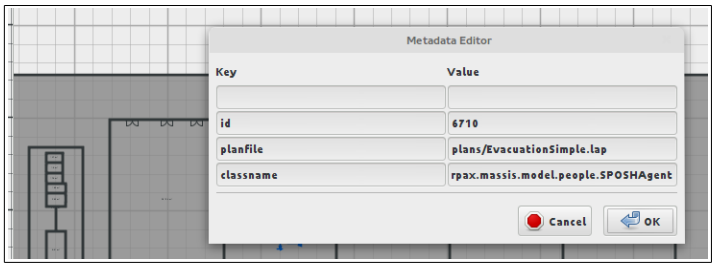

Figure 48: Metadata Editor

MASSIS needs extra, user-provided information about the elements of the building. This information is stored as metadata inside each element of the building. Every element in the building has metadata, can be viewed and edited directly from Tools → Add Metadata. This popups a simple modal dialog, with input fields in the form of key – value pairs. These values are read and processed later by MASSIS. Figure 48 shows an example of this editor.

4.2.4 Teleport Linking

In a building of multiple floors, the agents must be capable of moving through the different floors of the building. This is done via *Teleports* : Special elements in the building that, as the name suggests, they teleport the agent from one location to another, and they are unidirectional. A correctly configured teleport consists on two elements, representing the origin area and the destination area. Any furniture object can be configured as teleport. However, MASSIS release comes with a group of 3D objects designed with this purpose, in order to make easier the design and recognition of them, as it is shown in Figure 49.

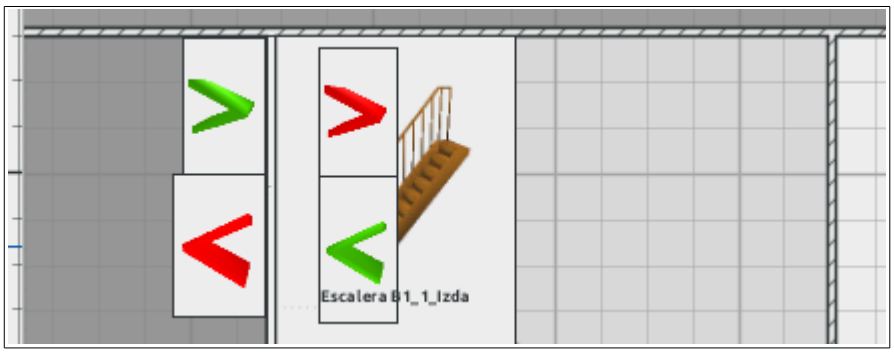

Figure 49: Teleports for emulating movement in stairs

A Teleport object must have the following parameters,

- `type` : `START` or `END`
- `teleport`: the teleport name

So, in the building, there would be two elements with the same teleport attribute, one with `START` as value of `type`, and the other one with `END` as value. Figure shows an example of an element conforming the Teleport `SB1_B_LEFT`.

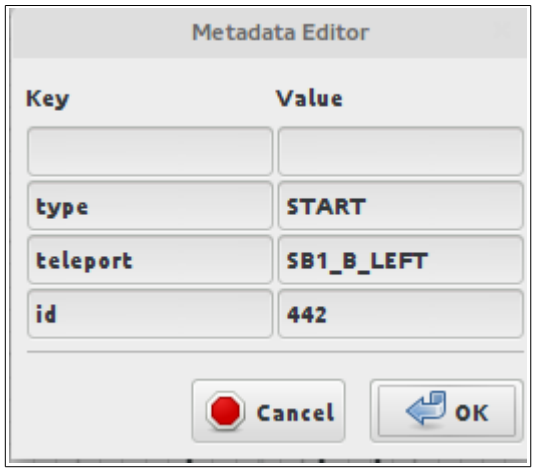

Figure 50: Teleport metadata

4.2.5 Other MASSIS' Design Utilities

MASSIS comes with some other design utilities, that can be found at *Tools* → *Designer tools* or *Name Generation* .

Designer tools

This plugin can make the *external* walls invisible(not every wall, so the interior of the building can be viewed from the outside, without making all the walls invisible).

One of the most boring things when designing a building is doing repetitive tasks one by one. For example, the color of the floor. This must be done clicking and modifying each room, in the SweetHome3D's way. With the *Rooms Color* plugin, this task can be done automatically. (Figure 51).

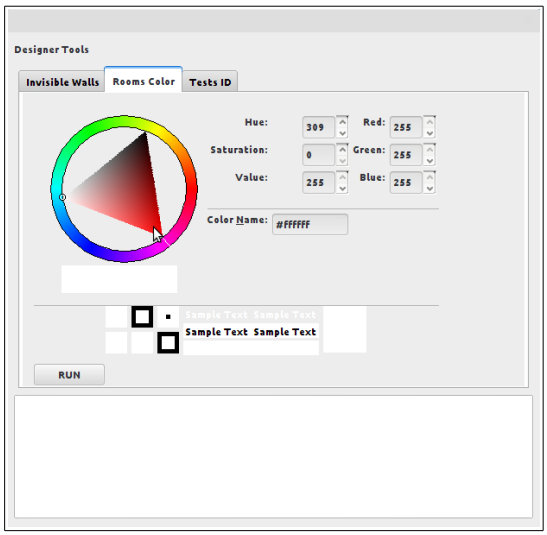

Figure 51: Rooms Color plugin

Name Generation

This plugin gives a name with the specified prefix to every room, or door in the building. (Figure 52)

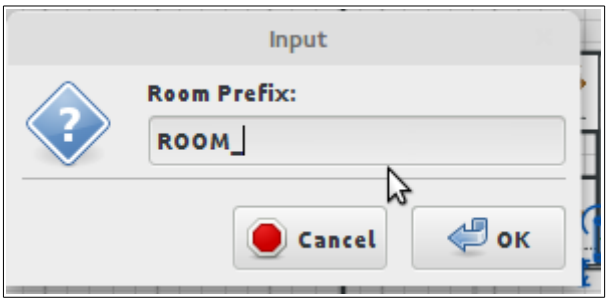

Figure 52: Name generation plugin

4.2.6 Final notes

This section is not intended to replace SweetHome3D's manual, only as a getting started guide. For more information, refer to the SweetHome3D manual. Can be found at `http://www.sweethome3d.-com/userGuide.jsp`.

4.3 Specifying Behaviors With Reactive Plans

4.3.1 Installing Netbeans and Pogamut's yaPOSH editor

Although POSH reactive plans can be designed *by hand*, it is much easier to do it graphically. Pogamut's plugins for Netbeans help to this task, integrating in the IDE a graphical plan editor.

Note: This is only a suggestion for a faster development of POSH Reactive plans, it is not necessary for running or extending MASSIS.

Downloading & installing Netbeans

The Pogamut's plugin interesting for MASSIS is the POSH plans graphical designer (yaPOSH editor). It is developed for the Netbeans platform (sorry, eclipse lovers!), so a recent copy of Netbeans (7.3+) should be obtained. This can be done through Netbeans download page [51]. Also, Java JDK 1.6+ is also needed (It can be downloaded bundled with Netbeans, check the options in the download page).

YaPOSH editor

The Pogamut's yaPOSH editor comes with the Pogamut UT2004 installer. This installer can be downloaded from Pogamut download page [52]. The latest stable version is recommended (Fig 53). The installation is fairly simple, but in case of any trouble the Pogamut's download page has detailed information about the installation steps.

Figure 53: Pogamut Platform download page

4.3.2 POSH Plan creation

Basic Behaviors Design

A good tutorial about how to create SPOSH plans with Pogamut can be found at :

`http://pogamut.cuni.cz/pogamut_files/latest/doc/tutorials/ch14s03.html`

4.3.3 Linking with SweetHome 3D & MASSIS

Using MASSIS metadata editor, the behavior of each agent can be specified in the fields `planfile` and `classname`. Figure 48 can serve as an example. The value of the `planfile` field must be relative, never absolute (As will be explained later, the location of the resources folder is a parameter of MASSIS launcher).

4.4 Simulation

4.4.1 Running a new simulation

For running a simulation, the parameter run-as must have as value `SIMULATOR`.

In addition, as every simulation (despite the type of it) meeds a building file, this should be provided also. The parameter `build-`

ing-path, tells MASSIS where is the building. The number of steps desired can be provided. If they are not, the simulation runs in an infinite loop.

The parameter simulation-mode should be SIMULATION.

For logging the simulation results into a file, the parameter save-simulation-to (optional) should contain a valid path for dumping the simulation timeline.

```
  java -jar MASSIS.jar --run-as=SIMULATOR -building-path \
<BUILDING_PATH> --run-for <STEPS> --simulation-mode SIMULA-
TION \
--save-simulation-to <SIMULATION_RESULTS_FILE>
```

The desired way of displaying the simulation progress (graphically or by console) should be specified with the display parameter (GUI or CONSOLE).

4.4.2 Loading a saved simulation

For playing a previous saved simulation, the value of the parameter simulation-mode must be PLAYBACK and also, the parameter load-simulation-from should be provided.

```
  java -jar MASSIS.jar --run-as=SIMULATOR -building-path \
<BUILDING_PATH> --run-for <STEPS> --simulation-mode
PLAYBACK \
--load-simulation-from <SIMULATION_RESULTS_FILE>
```

4.4.3 Help

The MASSIS launcher contains a help with a brief explanation of these parameters mentioned before. Calling MASSIS with the argument -help shows all the parameters available and their explanation.

5 Concluding Remarks

It's more fun to arrive a conclusion than to justify it.
Malcolm Forbes

5.1 Conclusion

This work has presented MASSIS, a multiagent-based simulation framework that supports the decision making process of humans when solving problems. This is achieved by simulating each agent individually, but with the support of several methods and efficient data structures that take advantage of particularities of the indoor domain.

The creation of the environments is done by SweetHome 3D, with some extensions for linking agent's behavior in the simulation. Agent behavior is structured in low-level and high-level behavior components, extending Pogamut's POSH implementation model, with the addition of features that facilitate the separation of decision making process and low level actions.

MASSIS provides a rich set of low-level behavior components for the simulation of indoor scenarios. This has required to MASSIS the extension of the SweetHome3D environment with plugins for linking agent's behavior in the simulation. In order to apply MASSIS to other kind of scenarios (e.g., a city), new low-level behavior components should be implemented and integrated with another graphical design package that supports the definition of the new environment.

In this sense, MASSIS can be easily extended. MASSIS provides as well a rich log capability, which can be the basis for further anal-

ysis of the scenarios. The extensibility of the MASSIS platform is well supported through its component based architecture. For instance, different visualizations can be managed during the simulation, new algorithms and agent attributes can be supported and monitored.

5.2 Future Work

MASSIS framework is only in its beginning, and we believe has huge potential. In the next version of the framework, there will be many updates to the framework. The most relevant are shown below.

Integration of existing analysis tools

One of the most relevant issues in the next version of MASSIS will be the integration of existing analysis tools, in order to make this framework more independent and useful.

Different AI behavior models

It is interesting to add new, different reasoning models, in order to test different approaches.

Using a more powerful 3D engine

Although SweetHome3D is good for this initial version of MASSIS, there are better alternatives (such as Jmonkey Engine [53]), with which more realistic results can be obtained.

Distributed Computing

Another feature that is being considered is making MASSIS capable to run over cluster and cloud-computing architectures. Since MASSIS runs over MASON, and there is a version of MASON designed to run in parallel (D-MASON [54]), it would be relatively easy to implement it, and it would add more value to MASSIS.

6 References

[1] E. Serrano and J. Botia, "Validating ambient intelligence based ubiquitous computing systems by means of artificial societies," *Information Sciences*, vol. 222, no. 0, pp. 3 – 24, 2013.

[2] D. Cook and S. Das, *Smart environments: Technology, protocols and applications*, vol. 43. John Wiley & Sons, 2004.

[3] P. Nixon, S. Dobson, and G. Lacey, "Managing Smart Environments," in *Proceedings of the Workshop on Software Engineering for Wearable and Pervasive Computing*, 2000.

[4] D. J. Cook, M. Youngblood, E. O. Heierman III, K. Gopalratnam, S. Rao, A. Litvin, and F. Khawaja, "MavHome: An agent-based smart home," in *2013 IEEE international conference on pervasive computing and communications (PerCom)*, 2003, pp. 521–521.

[5] J. C. Augusto and C. D. Nugent, *Designing smart homes: the role of artificial intelligence*, vol. 4008. Springer Science & Business Media, 2006.

[6] S. K. Das and D. J. Cook, "Designing smart environments: A paradigm based on learning and prediction," in *Pattern Recognition and Machine Intelligence*, Springer, 2005, pp. 80–90.

[7] A. Omicini, A. Ricci, and G. Vizzari, "Building smart environments as agent workspaces," in *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2007. WETICE 2007. 16th IEEE International Workshops on*, 2007, pp. 92–97.

[8] D. Hiebeler, "The swarm simulation system and individual-based modeling," 1994.

[9] N. Collier, "Repast: An extensible framework for agent simulation," *The University of Chicago's Social Science Research*, vol. 36, p. 2003, 2003.

[10] M. J. North, N. T. Collier, J. Ozik, E. R. Tatara, C. M. Macal, M. Bragen, and P. Sydelko, "Complex adaptive systems modeling with repast simphony," *Complex Adaptive Systems Modeling*, vol. 1, no. 1, pp. 1–26, 2013.

[11] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan, "Mason: A multiagent simulation environment," *Simulation*, vol. 81, no. 7, pp. 517–527, 2005.

[12] U. Wilensky, "NETLOGO itself: NetLogo," *Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, http://ccl. northwestern. edu/netlogo*, 1999.

[13] Legion, "Science in Motion." [Online]. Available: http://www.legion.com.

[14] M. Owen, E. R. Galea, and P. J. Lawrence, "The EXODUS evacuation model applied to building evacuation scenarios," *Journal of Fire Protection Engineering*, vol. 8, no. 2, pp. 65–84, 1996.

[15] PedGo, "TraffGo HT.," 2006. [Online]. Available: http://www.traffgo-ht.com/de/pedestrians/products/pedgo/index.html.

[16] M. MacDonald, "STEPS," 2009. [Online]. Available: http://www.steps.mottmac.com/.

[17] Thunderhead Engineering, "Pathfinder," 2006. [Online]. Available: http://www.thunderheadeng.com/pathfinder/.

[18] Golaem, "Golaem Crowd: Artist-Driven Crowd Simulation," 2011. [Online]. Available: http://www.thunderheadeng.com/pathfinder/.

[19] M. Schuerman, S. Singh, M. Kapadia, and P. Faloutsos, "Situation agents: agent-based externalized steering logic," *Computer Animation and Virtual Worlds*, vol. 21, no. 3–4, pp. 267–276, 2010.

[20] L. Saîfi, A. Boubetra, and F. Nouioua, "Approaches to Modeling the Emotional Aspects of a Crowd," in *Modelling and Simulation (EUROSIM), 2013 8th EUROSIM Congress on*, 2013, pp. 151–154.

[21] T. Bosse, M. Hoogendoorn, M. C. Klein, J. Treur, C. N. Van Der Wal, and A. Van Wissen, "Modelling collective decision making in groups and crowds: Integrating social contagion and interacting emotions, beliefs and intentions," *Autonomous Agents and Multi-Agent Systems*, vol. 27, no. 1, pp. 52–84, 2013.

[22] Massive Software, "Simulating Life," 2002. [Online]. Available: http://www.massivesoftware.com/.

[23] S. Wu and Q. Sun, "Computer simulation of leadership, consensus decision making and collective behaviour in humans," *PloS one*, vol. 9, no. 1, 2014.

[24] A. C. Bicharra, N. Sánchez-Pi, L. Correia, and J. M. Molina, "Multi-agent simulations for emergency situations in an airport scenario," *ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal*, vol. 1, no. 3, pp. 69–73, 2013.

[25] R. Hocevar, F. Marson, V. Cassol, H. Braun, R. Bidarra, and S. R. Musse, "From their environment to their behavior: a procedural approach to model groups of virtual agents," in *Intelligent Virtual Agents*, 2012, pp. 370–376.

[26] E. Puybaret, "Sweet Home 3D," 2005. [Online]. Available: http://www.sweethome3d.com/.

[27] J. Gemrot, R. Kadlec, M. Bída, O. Burkert, R. Píbil, J. Havlíček, L. Zemčák, J. Šimlovič, R. Vansa, M. Štolba, and others, "Pogamut 3 can assist developers in building AI (Not only) for their videogame agents," *Agents for Games and Simulations*, pp. 1–15, 2009.

[28] Epic Games, "Unreal Tournament 2004." [Online]. Available: http://liandri.beyondunreal.com/Unreal_Tournament_2004.

[29] Epic Games, "UE2 Runtime." [Online]. Available: http://wiki.beyondunreal.com/Unreal_Engine_2_Runtime.

[30] Epic Games, "UDK Documentation." [Online]. Available: https://www.unrealengine.com/previous-versions/documentation.

[31] Introversion Software, "Defcon: Everybody Dies." [Online]. Available: http://www.introversion.co.uk/defcon/.

[32] K. Woodward, "straightedge - 2D polygon library for games." [Online]. Available: https://code.google.com/p/straightedge/.

[33] V. Solutions, "JTS Topology Suite," 2015. [Online]. Available: http://www.vividsolutions.com/jts/JTSHome.htm.

[34] A. Treuille, S. Cooper, and Z. Popović, "Continuum crowds," *ACM Transactions on Graphics (TOG)*, vol. 25, no. 3, pp. 1160–1168, 2006.

[35] J. Shopf and A. GPG, "Crowd Simulation in Froblins."

[36] J. Moersch and H. Hamilton, "Hybrid Vector Field Pathfinding."

[37] R. A. Finkel and J. L. Bentley, "Quad trees a data structure for retrieval on composite keys," *Acta informatica*, vol. 4, no. 1, pp. 1–9, 1974.

[38] C. W. Reynolds, *Steering behaviors for autonomous characters*, vol. 1999. 1999.

[39] J. Orkin, "Symbolic representation of game world state: Toward real-time planning in games," in *Proceedings of the AAAI Workshop on Challenges in Game Artificial Intelligence*, 2004, vol. 5.

[40] J. Orkin, "Agent Architecture Considerations for Real-Time Planning in Games.," in *AIIDE*, 2005, pp. 105–110.

[41] Monolith Productions, "F.E.A.R.," 2005. [Online]. Available: http://en.wikipedia.org/wiki/F.E.A.R.

[42] R. E. Fikes and N. J. Nilsson, "STRIPS: A new approach to the application of theorem proving to problem solving," *Artificial intelligence*, vol. 2, no. 3, pp. 189–208, 1972.

[43] J. J. Bryson, "The behavior-oriented design of modular agent intelligence," in *Agent technologies, infrastructures, tools, and applications for e-services*, Springer, 2003, pp. 61–76.

[44] J. J. Bryson, "Intelligence by design: principles of modularity and coordination for engineering complex adaptive agents," 2001.

[45] R. A. Brooks, "A robust layered control system for a mobile robot," *Robotics and Automation, IEEE Journal of*, vol. 2, no. 1, pp. 14–23, 1986.

[46] R. A. Brooks., "Intelligence without reason," *The artificial life route to artificial intelligence: Building embodied, situated agents*, pp. 25–81, 1995.

[47] Ecma-international.org, "Standard ECMA-404," 2015. [Online]. Available: http://www.ecma-international.org/publications/standards/Ecma-404.htm.

[48] Google, "google-gson - A Java library to convert JSON to Java objects and vice-versa - Google Project Hosting." [Online]. Available: https://code.google.com/p/google-gson/.

[49] Julio Viera, "julman99/gson-fire," 2015. [Online]. Available: https://github.com/julman99/gson-fire.

[50] Sqlite.org, "SQLite Home Page," 2015. [Online]. Available: https://www.sqlite.org/.

[51] netbeans.org, "Netbeans download page," 2015. [Online]. Available: https://netbeans.org/downloads/index.html.

[52] P. Platform, "Pogamut Platform download page," 2015. [Online]. Available: http://diana.ms.mff.cuni.cz/main/tiki-index.php?page=Download.

[53] M. Powell, "JMonkey Engine," *Avaliable in: http://www. jmonkeyengine. com*, 2008.

[54] G. Cordasco, R. De Chiara, A. Mancuso, D. Mazzeo, V. Scarano, and C. Spagnuolo, "A framework for distributing agent-based simulations," in *Euro-Par 2011: Parallel Processing Workshops*, 2012, pp. 460–470.

7 Appendices

Appendix I: Rafael Pax, Juan Pavón: Agent-based Simulation of Crowds in Indoor Scenarios. 9th International Symposium on Intelligent Distributed Computing (IDC'2015), Guimaraes (Portugal), 7-9 oct. 2015 (accepted for oral presentation and full publication)

E-mail of the notification acceptance:

```
Subject: IDC'2015 notification for paper 20
From: "IDC'2015" <idc2015@easychair.org>
Date: 05/22/2015 05:49 PM
To: Rafael Pax <rpax@ucm.es>


Dear Rafael Pax,

We are glad to inform that your submission

Agent-based Simulation of Crowds in Indoor Scenarios

was accepted for Oral Presentation at the 9th International Symposium on
Intelligent Distributed Computing (IDC'2015) and for publication as a Full
Regular Paper (max 10 pages) in the IDC'2015 Conference Proceedings. You can
consult the detailed reviews at the end of this message.

The Conference Proceedings will be edited by Springer in a volume of the
Series Studies in Computational Intelligence.
In a few days we will send you a separate message with instructions on
how to prepare and submit the camera-ready version of your paper. The
deadline for submitting is June 19th, 2015.

All accepted papers must be presented orally at the Conference.

At least one author for each accepted paper must register for the
Conference. The strict deadline for author registration is June 19th, 2015.

On behalf of the IDC'2015 organization team, thank you for submitting
your work. We look forward to seeing you in Guimarães next October.

Best regards,

David Camacho
Paulo Novais
```

Appendix II: Rafael Pax, Juan Pavón: Multi-agent system simulation of InDoor Scenarios. 9th International Workshop on Multi-Agent Systems and Simulation (MAS&S'15). Lodz, Poland, September 13-16, 2015 (submitted, waiting for acceptance)